A Language for Generic Programming in the Large

Jeremy G. Siek [a]and Andrew Lumsdaine [b]

[a]Deptartment of Electrical and Computer Engineering, University of Colorado at Boulder, USA
[b]Computer Science Department, Indiana University, USA

Abstract

Generic programming is an effective methodology for developing reusable software libraries. Many programming languages provide generics and have features for describing interfaces, but none completely support the idioms used in generic programming. To address this need we developed the language $\mathcal{G}$. The central feature of $\mathcal{G}$ is the `concept`, a mechanism for organizing constraints on generics that is inspired by the needs of modern C++ libraries. $\mathcal{G}$ provides modular type checking and separate compilation (even of generics). These characteristics support modular software development, especially the smooth integration of independently developed components. In this article we present the rationale for the design of $\mathcal{G}$ and demonstrate the expressiveness of $\mathcal{G}$ with two case studies: porting the Standard Template Library and the Boost Graph Library from C++ to $\mathcal{G}$. The design of $\mathcal{G}$ shares much in common with the `concept` extension proposed for the next C++ Standard (the authors participated in its design) but there are important differences described in this article.

Key words: programming language design, generic programming, generics, polymorphism, concepts, associated types, software reuse, type classes, modules, signatures, functors, virtual types

1. Introduction

The 1968 NATO Conference on Software Engineering identified a software crisis affecting large systems such as IBM's OS/360 and the SABRE airline reservation system [1, 2]. At this conference McIlroy gave an invited talk entitled *Mass-produced Software Components* [3] proposing the systematic creation of reusable software components as a solution to the software crisis. He observed that most software is created from similar building blocks, so programmer productivity would be increased if a standard set of blocks could be shared among many software products. We are beginning to see the benefits of software reuse; Douglas McIlroy's vision is gradually becoming a reality. The number of commercial and open source software libraries

Email addresses: `jeremy.siek@colorado.edu` (Jeremy G. Siek), `lums@osl.iu.edu` (Andrew Lumsdaine).

is steadily growing and application builders often turn to libraries for user-interface components, database access, report creation, numerical routines, and network communication, to name a few. Furthermore, larger software companies have benefited from the creation of in-house domain-specific libraries which they use to support entire software product lines [4].

As the field of software engineering progresses, we learn better techniques for building reusable software. In the 1980s Musser and Stepanov developed a methodology for creating highly reusable algorithm libraries [5, 6, 7, 8], using the term *generic programming* for their work. [1] Their approach was novel in that they wrote algorithms not in terms of particular data structures but rather in terms of abstract requirements on structures based on the needs of the algorithm. Such generic algorithms could operate on any data structure provided that it meet the specified requirements. Preliminary versions of their generic algorithms were implemented in Scheme, Ada, and C. In the early 1990s Stepanov and Musser took advantage of the template feature of C++ [9] to construct the Standard Template Library (STL) [10, 11]. The STL became part of the C++ Standard, which brought generic programming into the mainstream. Since then, the methodology has been successfully applied to the creation of libraries in numerous domains [12, 13, 14, 15, 16].

The ease with which programmers develop and use generic libraries varies greatly depending on the language features available for expressing polymorphism and requirements on type parameters. In 2003 we performed a comparative study of modern language support for generic programming [17]. The initial study included C++, SML, Haskel, Eiffel, Java, and C#, and we evaluated the languages by porting a representative subset of the Boost Graph Library [13] to each of them. We recently updated the study to include OCaml and Cecil [18]. While some languages performed quite well, none were ideal for generic programming.

Unsatisfied with the state of the art, we began to investigate how to improve language support for generic programming. In general we wanted a language that could express the idioms of generic programming while also providing *modular type checking* and *separate compilation*. In the context of generics, modular type checking means that a generic function or class can be type checked independently of any instantiation and that the type check guarantees that any well-typed instantiation will produce well-typed code. Separate compilation is the ability to compile a generic function to native assembly code that can be linked into an application in constant time.

Our desire for modular type checking was a reaction to serious problems that plague the development and use of C++ template libraries. A C++ template definition is not type checked until after it is instantiated, making templates difficult to validate in isolation. Even worse, clients of template libraries are exposed to confusing error messages when they accidentally misuse the library. For example, the following code tries to use `stable_sort` with the iterators from the `list` class.

```
std::list<int> l;
std::stable_sort(l.begin(), l.end());
```

Fig. 1 shows a portion of the error message from GNU C++. The error message includes functions and types that the client should not have to know about such as `__inplace_stable_sort` and `_List_iterator`. It is not clear from the error message who is responsible for the error. The error message points inside the STL so the client might conclude that there is an error in the STL. This problem is not specific to the GNU C++ implementation, but is instead a symptom of the delayed type checking mandated by the C++ language definition.

[1] The term generic programming is often used to mean any use of generics, i.e., any use of parametric polymorphism or templates. The term is also used in the functional programming community for function generation based on algebraic datatypes, i.e., polytypic programming. Here, we use generic programming solely in the sense of Musser and Stepanov.

```
stl_algo.h: In function 'void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]':
stl_algo.h:2565:   instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2345: error: no match for 'std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&' operator
stl_algo.h:2565:   instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
   [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stable_sort_error.cpp:5:   instantiated from here
stl_algo.h:2349: error: no match for 'std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&' operator
stl_algo.h:2352: error: no match for 'std::_List_iterator<int, int&, int*>&  std::_List_iterator<int, int&, int*>&' operator
```

Fig. 1. A portion of the error message from a misuse of `stable_sort`.

Our desire for separate compilation was driven by the increasingly long compile times we (and others) were experiencing when composing sophisticated template libraries. With C++ templates, the compilation time of an application is a function of the amount of code in the application plus the amount of code in all template libraries used by the application (both directly and indirectly). We would much prefer a scenario where generic libraries can be separately compiled so that the compilation time of an application is just a function of the amount of code in the application.

With these desiderata in hand we began laying the theoretical groundwork by developing the calculus F^G [19]. F^G is based on System F [20, 21], the standard calculus for parametric polymorphism, and like System F, F^G has a modular type checker and can be separately compiled. In addition, F^G provides features for precisely expressing constraints on generics, introducing the `concept` feature with support for associated types and same-type constraints. The main technical challenge overcome in F^G is dealing with type equality inside of generic functions. One of the key design choices in F^G is that models are lexically scoped, making F^G more modular than Haskell in this regard. (We discuss this in more detail in Section 3.6.1.) Concurrently with our work on F^G, Chakravarty, Keller, and Peyton Jones responded to our comparative study by developing an extension to Haskell to support associated types [22, 23].

The next step after F^G was to add two more features needed to express generic libraries: concept-based overloading (used for algorithm specialization) and implicit argument deduction. Fully general implicit argument deduction is non-trivial in the presence of first-class polymophism (which is present in $\mathcal{G}$), but some mild restrictions make the problem tractable (Section 3.5). However, we discovered a a deep tension between concept-based overloading and separate compilation [24]. At this point our work bifurcated into two language designs: the language $\mathcal{G}$ which supports separate compilation and only a basic form of concept-based overloading [25, 26], and the concepts extension to C++ [27], which provides full support for concept-based overloading but not separate compilation. For the next revision of the C++ Standard, popularly referred to as C++0X, separate compilation for templates was not practical because the language already included template specialization, a feature that is also deeply incompatible with separate compilation. Thus, for C++0X it made sense to provide full support for concept-based overloading. For $\mathcal{G}$ we placed separate compilation as a higher priority, leaving out template specialization and requiring programmers to work around the lack of full concept-based overloading (see Section X).

Table 1 shows the results of our comparative study of language support for generic programming [18] augmented with new columns for C++0X and $\mathcal{G}$ and augmented with three new rows: modular type checking (previously part of "separate compilation"), lexically scoped models, and concept-based overloading. Table 2 gives a brief description of the evaluation criteria.

The rest of this article describes the design of $\mathcal{G}$ in detail. We review the essential ideas of generic programming and survey of the idioms used in the Standard Template Library (Sec-

Table 1
The level of support for generic programming in several languages. A black circle indicates full support for the feature or characteristic whereas a white circle indices lack of support. The rating of "-" in the C++ column indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported.

	C++	SML	OCaml	Haskell	Java	C#	Cecil	C++0X	$\mathcal{G}$
Multi-type concepts	-	●	○	●*	○	○	◒	●	●
Multiple constraints	-	◒	◒	●	●	●	●	●	●
Associated type access	●	●	◒	●†	◒	◒	◒	●	●
Constraints on assoc. types	-	●	●	●†	◒	◒	●	●	●
Retroactive modeling	-	●	●	●	○	○	●	●	●
Type aliases	●	●	●	●	○	○	○	●	●
Separate compilation	○	●	●	●	●	●	◒	○	●
Implicit arg. deduction	●	○	●	●	●	●	◒	●	●
Modular type checking	○	●	◒	●	●	●	◒	◒	●
Lexically scoped models	○	●	○	○	○	○	○	○	●
Concept-based overloading	●	○	○	○	○	○	●	●	◒

*Using the multi-parameter type class extension to Haskell [28].
†Using the proposed associated types extension to Haskell [23].

Table 2
Glossary of Evaluation Criteria

Criterion	Definition
Multi-type concepts	Multiple types can be simultaneously constrained.
Multiple constraints	More than one constraint can be placed on a type parameter.
Associated type access	Types can be mapped to other types within the context of a generic function.
Constraints on associated types	Concepts may include constraints on associated types.
Retroactive modeling	The ability to add new modeling relationships after a type has been defined.
Type aliases	A mechanism for creating shorter names for types is provided.
Separate compilation	Generic functions can be compiled independently of calls to them.
Implicit argument deduction	The arguments for the type parameters of a generic function can be deduced and do not need to be explicitly provided by the programmer.
Modular type checking	Generic functions can be compiled independently of calls to them.
Lexically scoped models	Model declarations are treated like any other declaration, and are in scope for the remainder of enclosing namespace. Models may be explicitly imported from other namespaces.
Concept-based overloading	There can be multiple generic functions with the same name but differing constraints. For a particular call, the most specific overload is chosen.

tion 2). This provides the motivation for the design of the language features in $\mathcal{G}$ (Section 3). We then evaluate $\mathcal{G}$ with respect to a port of the Standard Template Library (Section 4) and the Boost Graph Library (Section 5). We conclude with a survey of related work (Section 6) and with the future directions for our work (Section 7).

This article is an updated and greatly extended version of [26], providing a more detailed

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

Fig. 2. Definition of Generic Programming from Jazayeri, Musser, and Loos[29]

rationale for the design of $\mathcal{G}$ and extending our previous comparative study to include $\mathcal{G}$ by evaluating a port of the Boost Graph Library to $\mathcal{G}$.

2. Generic Programming and the STL

Fig. 2 reproduces the standard definition of generic programming from Jazayeri, Musser, and Loos [29]. The generic programming methodology always consists of the following steps: 1) identify a family of useful and efficient concrete algorithms with some commonality, 2) resolve the differences by forming higher-level abstractions, and 3) lift the concrete algorithms so they operate on these new abstractions. When applicable, there is a fourth step to implement automatic selection of the best algorithm, as described in Fig. 2.

2.1. *Type requirements, concepts, and models*

The `merge` algorithm from the STL, shown in Fig. 3, serves as a good example of generic programming. The algorithm does not directly work on a particular data structure, such as an array or linked list, but instead operates on an abstract entity, a concept. A *concept* is a collection of requirements on a type, or to look at it a different way, it is the set of all types that satisfy the requirements. For example, the *Input Iterator* concept requires that the type have an increment and dereference operation, and that both are constant-time operations. (We italicize concept names.) A type that meets the requirements is said to *model* the concept. (It helps to read "models" as "implements".) For example, the models of the *Input Iterator* concept include the built-in pointer types, such as `int*`, the iterator type for the `std::list` class, and the `istream_iterator` adaptor. Constraints on type parameters are primarily expressed by requiring the corresponding type arguments to model certain concepts. In the `merge` template, the argument for `InIter1` is required to model the *Input Iterator* concept. Type requirements are not expressible in C++, so the convention is to specify type requirements in comments or documentation as in Fig. 3.

Fig. 3. The `merge` algorithm in C++.

```
template<typename InIter1, typename InIter2, typename OutIter>
  // where InIter1 models Input Iterator, InIter2 models Input Iterator.
  //    OutIter models Output Iterator, writing the value_type of InIter1.
  //    The value_type of InIter1 and InIter2 are the same type.
  //    The value_type of InIter1 is Less Than Comparable.
OutIter merge(InIter1 first1, InIter1 last1,
              InIter2 first2, InIter2 last2, OutIter result) {
  while (first1 != last1 && first2 != last2) {
    if (*first2 < *first1) {
      *result = *first2; ++first2;
    } else {
      *result = *first1; ++first1;
    }
    ++result;
  }
  return copy(first2, last2, copy(first1, last1, result));
}
```

The type requirements for `merge` refer to relationships between types, such as the `value_type` of `InIter1`. This is an example of an *associated type*, which maps between types that are part of a concept. The `merge` algorithm also needs to express that the `value_type` of `InIter1` and `InIter2` are the same, which we call *same-type constraints*. Furthermore, the `merge` algorithm includes an example of how associated types and modeling requirements can be combined: the `value_type` of the input iterators is required to be *Less Than Comparable*.

Fig. 4 shows the definition of the *Input Iterator* concept following the presentation style used in the SGI STL documentation [30, 31]. In the description, the variable `X` is used as a place holder for the modeling type. The *Input Iterator* concept requires several associated types: `value_type`, `difference_type`, and `iterator_category`. Associated types change from model to model. For example, the associated `value_type` for `int*` is `int` and the associated `value_type` for `list<char>::iterator` is `char`. The *Input Iterator* concept requires that the associated types be accessible via the `iterator_traits` class. (Traits classes are discussed in Section 2.4). The `count` algorithm, which computes the number of occurrences of a value within a sequence, is a simple example for the need of this access mechanism, for it needs to access the `difference_type` to specify its return type:

```
template<typename Iter, typename T>
typename iterator_traits<Iter>::difference_type
count(Iter first, Iter last, const T& value);
```

The reason that `count` uses the iterator-specific `difference_type` instead of `int` is to accommodate iterators that traverse sequences that may be too long to be measured with an `int`.

In general, a concept may consist of the following kinds of requirements.

refinements are analogous to inheritance. They allow one concept to include the requirements from another concept.

operations specify the functions that must be implemented for the modeling type.

associated types specify mappings between types, and in C++ are provided using traits classes, which we discuss in Section 2.4.

nested requirements include requirements on associated types such as modeling a certain concept or being the same-type as another type. For example, the *Input Iterator* concept requires that the associated `difference_type` be a signed integral type.

semantic invariants specify behavioral expectations about the modeling type.

complexity guarantees specify constraints on how much time or space may be used by an operation.

2.2. *Overview of the STL*

The high-level structure of the STL is shown in Fig. 5. The STL contains over fifty generic algorithms and 18 container classes. The generic algorithms are implemented in terms of a family of iterator concepts, and the containers each provide iterators that model the appropriate iterator concepts. As a result, the STL algorithms may be used with any of the STL containers. In fact, the STL algorithms may be used with any data structure that exports iterators with the required capabilities.

Fig. 6 shows the hierarchy of STL's iterator concepts. An arrow indicates that the source concept is a refinement of the target. The iterator concepts arose from the requirements of algorithms: the need to express the minimal requirements for each algorithm. For example, the `merge` algorithm passes through a sequence once, so it only requires the basic requirements of *Input Iterator* for the two ranges it reads from and *Output Iterator* for the range it writes to. The `search` algorithm, which finds occurrences of a particular subsequence within a larger sequence, must make multiple passes through the sequence so it requires *Forward Iterator*. The `inplace_merge` algorithm needs to move backwards and forwards through the sequence, so it requires *Bidirectional Iterator*. And finally, the `sort` algorithm needs to jump arbitrary distances within the sequence, so it requires *Random Access Iterator*. (The `sort` function uses the introsort algorithm [32] which is partly based on quicksort [33].) Grouping type requirements into concepts enables significant reuse of these specifications: the *Input Iterator* concept is directly used as a type requirement in over 28 of the STL algorithms. The *Forward Iterator*, which refines *Input Iterator*, is used in the specification of over 22 STL algorithms.

The STL includes a handful of common data structures. When one of these data structures does not fulfill some specialized purpose, the programmer is encouraged to implement the appropriate specialized data structure. All of the STL algorithms can then be made available for the new data structure at the small cost of implementing iterators.

Many of the STL algorithms are higher-order: they take functions as parameters, allowing the user to customize the algorithm to their own needs. The STL defines over 25 function objects for creating and composing functions.

The STL also contains a collection of adaptor classes, which are parameterized classes that implement some concept in terms of the type parameter (which is the adapted type). For example, the `back_insert_iterator` adaptor implements *Output Iterator* in terms of any model of *Back Insertion Sequence*. The generic `copy` algorithm can then be used with `back_insert_iterator` to append some integers to a list. Adaptors play an important role in the plug-and-play nature of the STL and enable a high degree of reuse.

Input Iterator

Description
An *Input Iterator* is an iterator that may be dereferenced to refer to some object, and that may be incremented to obtain the next iterator in a sequence. *Input Iterators* are not required to be mutable. The underlying sequence elements is not required to be persistent. For example, an *Input Iterator* could be reading input from the terminal. Thus, an algorithm may not make multiple passes through a sequence using an *Input Iterator*.

Refinement of
Trivial Iterator.

Notation

`X`	A type that is a model of *Input Iterator*
`T`	The value type of `X`
`i, j`	Objects of type `X`
`t`	Object of type `T`

Associated types

`iterator_traits<X>::value_type` The type of the value obtained by dereferencing an *Input Iterator*
`iterator_traits<X>::difference_type` A signed integral type used to represent the distance from one iterator to another, or the number of elements in a range.
`iterator_traits<X>::iterator_category` A type convertible to `input_iterator_tag`.

Definitions
An iterator is *past-the-end* if it points beyond the last element of a container. Past-the-end values are nonsingular and nondereferenceable. An iterator is *valid* if it is dereferenceable or past-the-end. An iterator `i` is *incrementable* if there is a ”next” iterator, that is, if `++i` is well-defined. Past-the-end iterators are not incrementable. An *Input Iterator* `j` is *reachable* from an *Input Iterator* `i` if, after applying `operator++` to `i` a finite number of times, `i == j`. The notation `[i,j)` refers to a range of iterators beginning with `i` and up to but not including `j`. The range `[i,j)` is a *valid range* if both `i` and `j` are valid iterators, and `j` is reachable from `i`.

Valid expressions
In addition to the expressions in *Trivial Iterator*, the following expressions must be valid.

expression	return type	semantics, pre/post-conditions
`*i`	Convertible to `T`	pre: `i` is incrementable
`++i`	`X&`	pre: `i` is dereferenceable, post: `i` is dereferenceable or past the end
`i++`		Equivalent to `(void)++i`.
`*i++`		Equivalent to `{T t = *i; ++i; return t;}`

Complexity guarantees
All operations are amortized constant time.

Models
`istream_iterator`, `int*`, `list<string>::iterator`, ...

Fig. 4. Documentation for the *Input Iterator* concept.

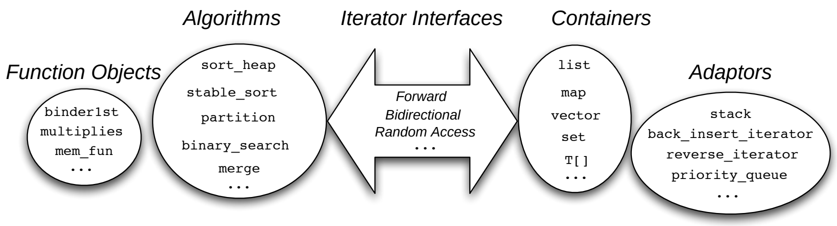

Fig. 5. High-level structure of the STL.

$$\begin{tikzcd}
\textbf{Random Access} \arrow[r] & \textbf{Bidirectional} \arrow[r] & \textbf{Forward} \arrow[r] \arrow[dr] & \textbf{Input} \\
 & & & \textbf{Output}
\end{tikzcd}$$

Fig. 6. The refinement hierarchy of iterator concepts.

2.3. *The problem of argument dependent name lookup in C++*

In C++, uses of names inside of a template definition, such as the use of `operator<` inside of `merge`, are resolved after instantiation. For example, when `merge` is instantiated with an iterator whose `value_type` is of type `foo::bar`, overload resolution looks for an `operator<` defined for `foo::bar`. If there is no such function defined in the scope of `merge`, the C++ compiler also searches the namespace where the arguments' types are defined, so looks for `operator<` in namespace `foo`. This rule is known as *argument dependent lookup* (ADL).

The combination of implicit instantiation and ADL makes it convenient to call generic functions. This is a nice improvement over passing concept operations as explicit arguments to a generic function, as in the `inc` example from Section 1. However, ADL has two flaws. The first problem is that the programmer calling the generic algorithm no longer has control over which functions are used to satisfy the concept operations. Suppose that namespace `foo` is a third party library and the application programmer writing the `main` function has defined his own `operator<` for `foo::bar`. ADL does not find this new `operator<`.

The second and more severe problem with ADL is that it opens a hole in the protection that namespaces are suppose to provide. ADL is applied uniformly to all name lookup, whether or not the name is associated with a concept in the type requirements of the template. Thus, it is possible for calls to helper functions to get hijacked by functions with the same name in other namespaces. Fig. 7 shows an example of how this can happen. The function template `lib::generic_fun` calls `load` with the intention of invoking `lib::load`. In `main` we call `generic_fun` with an object of type `foo::bar`, so in the call to `load`, `x` also has type `foo::bar`. Thus, argument dependent lookup also consider namespace `foo` when searching for `load`. There happens to be a function named `load` in namespace `foo`, and it is a slightly better match than `lib::foo`, so it is called instead, thereby hijacking the call to `load`.

Fig. 7. Example problem caused by ADL.

```
namespace lib {
  template<typename T> void load(T x, string)
    { std::cout << "Proceeding as normal!\n"; }
  template<typename T> void generic_fun(T x)
    { load(x, "file"); }
}
namespace foo {
  struct bar { int n; };
  template<typename T> void load(T x, const char*)
    { std::cout << "Hijacked!\n"; }
}
int main() {
  foo::bar a;
  lib::generic_fun(a);
}
// Output: Hijacked!
```

2.4. *Traits classes, template specialization, and separate type checking*

The traits class idiom plays an important role in writing generic algorithms in C++. Unfortunately there is a deep incompatibility between the underlying language feature, template specialization, and our goal of separate type checking.

A traits class [34] maps from a type to other types or functions. Traits classes rely on C++ template specialization to perform this mapping. For example, the following is the primary template definition for `iterator_traits`.

```
template<typename Iterator>
struct iterator_traits { ... };
```

A specialization of `iterator_traits` is defined by specifying particular type arguments for the template parameter and by specifying an alternate body for the template. The code below shows a user-defined iterator class, named `my_iter`, and a specialization of `iterator_traits` for `my_iter`.

```
class my_iter {
  float operator*() { ... }
  ...
};
template<> struct iterator_traits<my_iter> {
  typedef float value_type;
  typedef int difference_type;
  typedef input_iterator_tag iterator_category;
};
```

When the type `iterator_traits<my_iter>` is used in other parts of the program it refers to the above specialization. In general, a template use refers to the most specific specialization that matches the template arguments, if there is one, or else it refers to an instantiation of the primary template definition.

The use of iterator_traits within a template (and template specialization) represents a problem for separate compilation. Consider how a compiler might type check the following unique_copy function template.

```
template<typename InIter, typename OutIter>
OutIter unique_copy(InIter first, InIter last, OutIter result) {
  typename iterator_traits<InIter>::value_type value = *first;
  // ...
}
```

To check the first line of the body, the compiler needs to know that the type of *first is the same type as (or at least convertible to) the value_type member of iterator_traits<InIter>. However, prior to instantiation, the compiler does not know what type InIter will be instantiated to, and which specialization of iterator_traits to choose (and different specializations may have different definitions of the value_type).

Thus, if we hope to provide modular type checking, we must develop and alternative to using traits classes for accessing associated types.

2.5. *Concept-based overloading using the tag dispatching idiom*

One of the main points in the definition of generic programming in Fig. 2 is that it is sometimes necessary to provide more than one generic algorithm for the same purpose. When this happens, the standard approach in C++ libraries is to provide automatic dispatching to the appropriate algorithm using the tag dispatching idiom or enable_if [35]. Fig. 8 shows the advance algorithm of the STL as it is typically implemented using the tag dispatching idiom. The advance algorithm moves an iterator forward (or backward) n positions. There are three overloads of advance_dispatch, each with an extra iterator tag parameter. The C++ Standard Library defines the following iterator tag classes, with their inheritance hierarchy mimicking the refinement hierarchy of the corresponding concepts.

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

The main advance function obtains the tag for the particular iterator from iterator_traits and then calls advance_dispatch. Normal static overload resolution then chooses the appropriate overload of advance_dispatch. Both the use of traits and the overload resolution rely on knowing actual argument types of the template and the late type checking of C++ templates. So the tag dispatching idiom provides another challenge for designing a language for generic programming with separate type checking.

2.6. *Reverse iterators and conditional models*

The reverse_iterator class template adapts a model of *Bidirectional Iterator* and implements *Bidirectional Iterator*, flipping the direction of traversal so operator++ goes backwards and operator-- goes forwards. An excerpt from the reverse_iterator class template is shown below.

Fig. 8. The `advance` algorithm and the tag dispatching idiom.

```
template<typename InIter, typename Distance>
void advance_dispatch(InIter& i, Distance n, input_iterator_tag) {
  while (n--) ++i;
}
template<typename BidirIter, typename Distance>
void advance_dispatch(BidirIter& i, Distance n,
                      bidirectional_iterator_tag) {
  if (n > 0) while (n--) ++i;
  else while (n++) --i;
}
template<typename RandIter, typename Distance>
void advance_dispatch(RandIter& i, Distance n,
                      random_access_iterator_tag) {
  i += n;
}
template<typename InIter, typename Distance>
void advance(InIter& i, Distance n) {
  typename iterator_traits<InIter>::iterator_category cat;
  advance_dispatch(i, n, cat);
}
```

```
template<typename Iter>
class reverse_iterator {
protected:
  Iter current;
public:
  explicit reverse_iterator(Iter x) : current(x) { }
  reference operator*() const { Iter tmp = current; return *--tmp; }
  reverse_iterator& operator++() { --current; return *this; }
  reverse_iterator& operator--() { ++current; return *this; }
  reverse_iterator operator+(difference_type n) const
    { return reverse_iterator(current - n); }
  ...
};
```

The `reverse_iterator` class template is an example of a type that models a concept conditionally: if `Iter` models *Random Access Iterator*, then so does `reverse_iterator<Iter>`. The definition of `reverse_iterator` defines all the operations, such as `operator+`, required of a *Random Access Iterator*. The implementations of these operations rely on the *Random Access Iterator* operations of the underlying `Iter`. One might wonder why `reverse_iterator` can be used on iterators such as `list<int>::iterator` that are bidirectional but not random access. The reason this works is that a member function such as `operator+` is type checked and compiled only if it is used. For $\mathcal{G}$ we need a different mechanism to handle this, since function definitions are always type checked.

2.7. *Summary of language requirements*

In this section we surveyed how generic programming is accomplished in C++, taking note of the variety of language features and idioms that are used in current practice. In this section we summarize the findings as a list of requirements for a language to support generic programming.

(i) The language provides type parameterized functions with the ability to express constraints on the type parameters. The definitions of parameterized functions are type checked independently of how they are instantiated.

(ii) The language provides a mechanism, such as "concepts", for naming and grouping requirements on types, and a mechanism for composing concepts (refinement).

(iii) Type requirements include:
- requirements for functions and parameterized functions
- associated types
- requirements on associated types
- same-type constraints

(iv) The language provides an implicit mechanism for providing type-specific operations to a generic function, but this mechanism should maintain modularity (in contrast to argument dependent lookup in C++).

(v) The language implicitly instantiates generic functions when they are used.

(vi) The language provides a mechanism for concept-based dispatching between algorithms.

(vii) The language provides function expressions and function parameters.

(viii) The language supports conditional modeling.

3. The Design of $\mathcal{G}$

$\mathcal{G}$ is a statically typed imperative language with syntax and memory model similar to C++. We have implemented a compiler that translates $\mathcal{G}$ to C++, but $\mathcal{G}$ could also be interpreted or compiled to byte-code. Compilation units are separately type checked and may be separately compiled, relying only on forward declarations from other compilation units (even compilation units containing generic functions and classes). The languages features of $\mathcal{G}$ that support generic programming are the following:
- Concept and model definitions, including associated types and same-type constraints;
- Constrained polymorphic functions, classes, structs, and type-safe unions;
- Implicit instantiation of polymorphic functions; and
- Concept-based function overloading.

In addition, $\mathcal{G}$ includes the basic types and control constructs C++.

3.1. *Concepts*

The following grammar defines the syntax for concepts.

```
decl ← concept cid<tyid,...> { cmem ... };
cmem ← funsig | fundef          // Required operations
      | type tyid;              // Associated types
      | type == type;           // Same type constraints
      | refines cid<type, ...>;
      | require cid<type, ...>;
```

Fig. 9. The definition of the *Input Iterator* concept in $\mathcal{G}$.

```
concept InputIterator<X> {
  type value;
  type difference;
  refines EqualityComparable<X>;
  refines Regular<X>; // Regular refines Assignable and CopyConstructible
  require SignedIntegral<difference>;
  fun operator*(X b) -> value@;
  fun operator++(X! c) -> X!;
};
```

The grammar variable *cid* is for concept names and *tyid* is for type variables. The type variables are place holders for the modeling type (or a list of types for multi-type concepts). *funsig* and *fundef* are function signatures and definitions, whose syntax we introduce later in this section. In a concept, a function signature says that a model must define a function with the specified signature. A function definition in a concept provides a default implementation.

The syntax `type` *tyid* ; declares an associated type; a model of the concept must provide a type definition for the given type name. The syntax *type* == *type* introduces a same type constraint. In the context of a model definition, the two type expressions must refer to the same type. When the concept is used in the type requirements of a polymorphic function or class, this type equality may be assumed. Type equality in $\mathcal{G}$ is non-trivial, and is explained in Section 3.9. Concepts may be composed with `refines` and `require`. The distinction is that refinement brings in the associated types from the “super” concept. Fig. 9 shows an example of a `concept` definition in $\mathcal{G}$, the definition of `InputIterator`.

3.2. *Models*

The modeling relation between a type and a concept is established with a model definition using the following syntax.

decl ← `model` [`<`*tyid*, ...`>`] [`where` `{` *constraint*`,` ... `}`] *cid*`<`*type*, ...`>` `{` *decl* ...`};`

The following shows an example of the *Monoid* concept and a model definition that makes `int` a model of *Monoid*, using addition for the binary operator and zero for the identity element.

```
concept Monoid<T> {
  fun identity_elt() -> T@;
  fun binary_op(T,T) -> T@;
};
model Monoid<int> {
  fun binary_op(int x, int y) -> int@ { return x + y; }
  fun identity_elt() -> int@ { return 0; }
};
```

A model definition must satisfy all requirements of the concept. Requirements for associated types are satisfied by type definitions. Requirements for operations may be satisfied by function definitions in the model, by the `where` clause, or by functions in the lexical scope preceding the

Fig. 10. `reverse_iterator` conditionally models the *Random Access Iterator* concept.

```
model <Iter> where { RandomAccessIterator<Iter> }
RandomAccessIterator< reverse_iterator<Iter> >
{
  fun operator+(reverse_iterator<Iter> r, difference n)
     -> reverse_iterator<Iter>@
    { return @reverse_iterator<Iter>(r.current + n); }
  fun operator-(reverse_iterator<Iter> r, difference n)
    -> reverse_iterator<Iter>@
    { return @reverse_iterator<Iter>(r.current - n); }
  fun operator-(reverse_iterator<Iter> a, reverse_iterator<Iter> b)
     -> difference
    { return a.current - b.current; }
};
```

model definition. Refinements and nested requirements are satisfied by preceding model definitions in the lexical scope or by the `where` clause.

A model may be parameterized by placing type variables inside <>'s after the `model` keyword. The following definition establishes that all pointer types are models of `InputIterator`.

```
model <T> InputIterator<T*> {
  type value = T;
  type difference = ptrdiff_t;
};
```

The optional `where` clause in a model definition can be used to introduce constraints on the type variables. Constraints are either modeling constraints or same-type constraints.

constraint ← *cid*`<`*type*`, ...>` | *type* `==` *type*

Using the `where` clause we can express conditional modeling. As mentioned in Section 2.6, we need conditional modeling to say that `reverse_iterator` is a model of *Random Access Iterator* whenever the underlying iterator is. Fig. 10 shows is a model definition that says just this.

The rules for type checking parameterized model definitions with constraints is essentially the same as for generic functions, which we discuss in Section 3.4.

3.3. *Nominal versus structural conformance*

One of the fundamental design choices of $\mathcal{G}$ was to include model definitions. After all, it is possible to instead have the compiler figure out when a type has implemented all of the requirements of a concept. We refer to the approach of using explicit model definitions *nominal conformance* whereas the implicit approach we call *structural conformance*. An example of the nominal versus structural distinction can be seen in the example below. Do the concepts create two ways to refer to the same concept or are they different concepts that happen to have the same constraints?

```
concept A<T> {
  fun foo(T x) -> T;
};
```

```
concept B<T> {
  fun foo(T x) -> T;
};
```

With nominal conformance, the above are two different concepts, whereas with structural conformance, `A` and `B` are two names for the same concept. Examples of language mechanisms providing nominal conformance include Java interfaces and Haskell type classes. Examples of language mechanisms providing structural conformance include ML signatures [36], Objective Caml object types [37], CLU type sets [38], and Cforall specifications [39].

Choosing between nominal and structural conformance is difficult because both options have good arguments in their favor.

Structural conformance is more convenient than nominal conformance With nominal conformance, the modeling relationship is established by an explicit declaration. For example, a Java class declares that it `implements` an interface. In Haskell, an `instance` declaration establishes the conformance between a particular type and a type class. When the compiler sees the explicit declaration, it checks whether the modeling type satisfies the requirements of the concept and, if so, adds the type and concept to the modeling relation.

Structural conformance, on the other hand, requires no explicit declarations. Instead, the compiler determines on a need-to-know basis whether a type models a concept. The advantage is that programmers need not spend time writing explicit declarations.

Nominal conformance is safer than structural conformance The usual argument against structural conformance is that it is prone to *accidental conformance*. The classic example of this is a cowboy object being passed to something expecting a `Window` [40]. The `Window` interface includes a `draw()` method, which the cowboy has, so the type system does not complain even though something wrong has happened. This is not a particularly strong argument because the programmer has to make a big mistake for this kind accidental conformance to occur.

However, the situation changes for languages that support concept-based overloading. For example, in Section 2.5 we discussed the tag-dispatching idiom used in C++ to select the best `advance` algorithm depending on whether the iterator type models *Random Access Iterator* or only *Input Iterator*. With concept-based overloading, it becomes possible for accidental conformance to occur without the programmer making a mistake. The following C++ code is an example where an error would occur if structural conformance were used instead of nominal.

```
std::vector<int> v;
std::istream_iterator<int> in(std::cin), in_end;
v.insert(v.begin(), in, in_end);
```

The `vector` class has two versions of `insert`, one for models of *Input Iterator* and one for models of *Forward Iterator*. An *Input Iterator* may be used to traverse a range only a single time, whereas a *Forward Iterator* may traverse through its range multiple times. Thus, the version of `insert` for *Input Iterator* must resize the vector multiple times as it progresses through the input range. In contrast, the version of `insert` for *Forward Iterator* is more efficient because it first discovers the length of the range (by calling `std::distance`, which traverses the input range), resizes the vector to the correct length, and then initializes the vector from the range.

The problem with the above code is that `istream_iterator` fulfills the syntactic requirements for a *Forward Iterator* but not the semantic requirements: it does not support multiple passes. That is, with structural conformance, there is a false positive and `insert` dispatches to the version for *Forward Iterator*s. The program resizes the vector to the appropriate size for all the input but it does not initialize the vector because all of the input has already been read.

Why not both? It is conceivable to provide both nominal and structural conformance on a concept-by-concept basis, which is in fact the approach used in the concept extension for C++0X. Concepts that are intended to be used for dispatching could be nominal and other concepts could be structural. This matches the current C++ practice: some concepts come with traits classes that provide nominal conformance whereas other concepts do not (the default situation with C++ templates is structural conformance). However, providing both nominal conformance and structural conformance complicates the language, especially for programmers new to the language, and degrades its uniformity. Therefore, with $\mathcal{G}$ we provide only nominal conformance, giving priority to safety and simplicity over convenience.

3.4. *Generic Functions*

The syntax for generic functions is shown below. The name of the function is the identifier after `fun`, the type parameters are between the `<>`'s and are constrained by the requirement in the `where` clause. A function's parameters are between the `()`'s and the return type of a function comes after the `->`.

```
fundef ← fun id [<tyid,...>] [where { constraint, ... }]
                 (type pass [id], ...) -> type pass { stmt ... }
funsig ← fun id [<tyid,...>] [where { constraint, ... }]
                (type pass [id], ...) -> type pass;
decl ← fundef | funsig
pass ← mut ref        // pass by reference
       | @             // pass by value
mut ← const | ε       // constant
       | !             // mutable
ref ← & | ε
```

The default parameter passing mode in $\mathcal{G}$ is read-only pass-by-reference. Read-write pass-by-reference is indicated by `!` and pass-by-value is indicated by `@`.

The `merge` algorithm, implemented as a generic function in $\mathcal{G}$, is shown in Fig. 11. The function is parameterized on three types: `Iter1`, `Iter2`, and `Iter3`. The dot notation is used to refer to a member of a model, including associated types such as the `value` type of an iterator.

```
assoc ← cid<type, ...>.id | cid<type, ...>.assoc
type ← assoc
```

The *Output Iterator* concept used in the `merge` function is an example of a multi-parameter concept. It has a type parameter `X` for the iterator and a type parameter `T` for the type that can be written to the iterator. The following is the definition of the *Output Iterator* concept.

```
concept OutputIterator<X,T> {
  refines Regular<X>;
  fun operator<<(X! c, T t) -> X!;
};
```

In general the body of a generic function contains a sequence of statements. Syntax for some of the statements in $\mathcal{G}$ is defined in the following grammar.

```
stmt ← let id = expr; | while (expr) stmt | return expr; | expr; | ...
```

Fig. 11. The merge algorithm in $\mathcal{G}$.

```
fun merge<Iter1,Iter2,Iter3>
where { InputIterator<Iter1>, InputIterator<Iter2>,
        LessThanComparable<InputIterator<Iter1>.value>,
        InputIterator<Iter1>.value == InputIterator<Iter2>.value,
        OutputIterator<Iter3, InputIterator<Iter1>.value> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
  -> Iter3@
{
  while (first1 != last1 and first2 != last2) {
    if (*first2 < *first1) {
      result << *first2; ++first2;
    } else {
      result << *first1; ++first1;
    }
  }
  return copy(first2, last2, copy(first1, last1, result));
}
```

The `let` form introduces local variables, deducing the type of the variable from the right-hand side expression (similar to the `auto` proposal for C++0X [41]).

The body of a generic function is type checked separately from any instantiation of the function. The type parameters are treated as abstract types so no type-specific operations may be applied to them unless otherwise specified by the `where` clause. The `where` clause introduces surrogate model definitions and function signatures (for all the required concept operations) into the scope of the function.

Multiple functions with the same name may be defined, and static overload resolution is performed by $\mathcal{G}$ to decide which function to invoke at a particular call site depending on the argument types and also depending on which model definitions are in scope. When more than one overload may be called, the most specific overload is called (if one exists) according to the rules described in Section 3.10.

3.5. *Function calls and implicit instantiation*

The syntax for calling functions (or polymorphic functions) is the C-style notation:

$expr \leftarrow expr\,(expr\,,\ \ldots)$

Arguments for the type parameters of a polymorphic function need not be supplied at the call site: $\mathcal{G}$ will deduce the type arguments by unifying the types of the arguments with the types of the parameters and then implicitly instantiate the polymorphic function. The design issues surrounding implicit instantiation are described below. All of the requirements in the `where` clause must be satisfied by model definitions in the lexical scope preceding the function call, as described in Section 3.6. The following is an example of calling the generic `accumulate` function. In this case, the generic function is implicitly instantiated with type argument `int*`.

```
fun main() -> int@@ {
    let a = new int[8];
    a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 4; a[4] = 5;
    let s = accumulate(a, a + 5);
    if (s == 15) return 0;
    else return -1;
  }
```

A polymorphic function may be explicitly instantiated using this syntax:

$expr \leftarrow expr$`<|`ty`, `$\ldots$`|>`

Following Mitchell [42] we view implicit instantiation as a kind of coercion that transforms an expression of one type to another type. In the example above, the `accumulate` function was coerced from

```
  fun <Iter> where
    { InputIterator<Iter>, Monoid<InputIterator<Iter>.value> }
    (Iter@, Iter) -> InputIterator<Iter>.value@
```

to

```
  fun (int*@, int*) -> InputIterator<int*>.value@
```

There are several kinds of implicit coercions in $\mathcal{G}$, and together they form a subtyping relation $\leq$. The subtyping relation is reflexive and transitive. Like C++, $\mathcal{G}$ contains some bidirectional implicit coercions, such as `float` $\leq$ `double` and `double` $\leq$ `float`, so $\leq$ is not anti-symmetric. The subtyping relation for $\mathcal{G}$ is defined by a set of subtyping rules. The following is the subtyping rule for generic function instantiation.

$$(\textsc{Inst})\frac{\Gamma \text{ satisfies } \overline{c}}{\Gamma \vdash \texttt{fun<}\overline{\alpha}\texttt{>where}\{\overline{c}\}\texttt{(}\overline{\sigma}\texttt{)->}\tau \leq [\overline{\rho}/\overline{\alpha}](\texttt{fun(}\overline{\sigma}\texttt{)->}\tau)}$$

The type parameters $\overline{\alpha}$ are substituted for type arguments $\overline{\rho}$ and the constraints in the `where` clause must be satisfied in the current environment. To apply this rule, the compiler must choose the type arguments. We call this *type argument deduction* and discuss it in more detail momentarily. Constraint satisfaction is discussed in Section 3.6.

The subtyping relation allows for coercions during type checking according to the subsumption rule:

$$(\textsc{Sub})\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash e : \tau}$$

The (Sub) rule is not syntax-directed so its addition to the type system would result in a non-deterministic type checking algorithm. The standard workaround is to omit the above rule and instead allow coercions in other rules of the type system such as the rule for function application. The following is a rule for function application that allows coercions in both the function type and in the argument types.

$$(\textsc{App})\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash \overline{e_2} : \overline{\sigma_2} \qquad \Gamma \vdash \tau_1 \leq \texttt{fun(}\overline{\sigma_3}\texttt{)->}\tau_2 \qquad \Gamma \vdash \overline{\sigma_2} \leq \overline{\sigma_3}}{\Gamma \vdash e_1(\overline{e_2}) : \tau_2}$$

As mentioned above, the type checker must guess the type arguments $\overline{\rho}$ to apply the (Inst) rule. In addition, the (App) rule includes several types that appear from nowhere: $\overline{\sigma_3}$ and $\overline{\tau_2}$. The problem of deducing these types is equivalent to trying to find solutions to a system of inequalities. Consider the following example program.

```
  fun apply<T>(fun(T)->T f, T x) -> T { return f(x); }
  fun id<U>(U a) -> U { return a; }
  fun main() -> int@ { return apply(id, 0); }
```

The application `apply(id, 0)` type checks if there is a solution to the following system:

```
fun<T>(fun(T)->T, T) -> T ≤ fun(α, β) -> γ
fun<U>(U)->U ≤ α
int ≤ β
```

The following type assignment is a solution to the above system.

```
α = fun(int)->int
β = int
γ = int
```

Unfortunately, not all systems of inequalities are as easy to solve as the above example. In fact, with Mitchell's original set of subtyping rules, the problem of solving systems of inequalities was proved undecidable by Tiuryn and Urzyczyn [43]. There are several approaches to dealing with this undecidability.

Remove the (ARROW) rule. Mitchell's subtyping relation included the usual co/contravariant rule for functions.

$$\text{(ARROW)}\ \frac{\overline{\sigma_2} \leq \overline{\sigma_1} \qquad \tau_1 \leq \tau_2}{\texttt{fun(}\overline{\sigma_1}\texttt{)->}\tau_1 \leq \texttt{fun(}\overline{\sigma_2}\texttt{)->}\tau_2}$$

The (ARROW) rule is nice to have because it allows a function to be coerced to a different type so long as the parameter and return types are coercible in the appropriate way. In the following example the standard `ilogb` function is passed to `foo` even though it does not match the expected type. The (ARROW) rule allows for this coercion because `int` is coercible to `double`.

```
include "math.h"; // fun ilogb(double x) > int;
fun foo(fun(int)->int@ f) -> int@ { return f(1); }
fun main() -> int@ { return foo(ilogb); }
```

However, the (ARROW) rule is one of the culprits in the undecidability of the subtyping problem; removing it makes the problem decidable [43]. The language ML^{F} of Le Botlan and Remy [44] takes this approach, and for the time being, so does $\mathcal{G}$. With this restriction, type argument deduction is reduced to the variation of unification defined in [44]. Instead of working on a set of variable assignments, this unification algorithm keeps track of either a type assignment or the tightest lower bound seen so far for each variable. The (APP) rule for $\mathcal{G}$ is reformulated as follows to use this `unify` algorithm.

$$\text{(APP)}\ \frac{\begin{array}{cc}\Gamma \vdash e_1 : \tau_1 & \Gamma \vdash \overline{e_2} : \overline{\sigma_2} \\ Q = \{\tau_1 \leq \alpha, \overline{\sigma_2} \leq \overline{\beta}\} & Q' = \texttt{unify}(\alpha, \texttt{fun(}\overline{\beta}\texttt{)->}\gamma, Q)\end{array}}{\Gamma \vdash e_1(\overline{e_2}) : Q'(\gamma)}$$

In languages where functions are often written in curried form, it is important to provide even more flexibility than in the above (APP) rule by postponing instantiation, as is done in ML^{F}. Consider the `apply` example again, but this time written in curried form.

```
fun apply<T>(fun(T)->T f) -> (fun(T)->T)@ {
  return fun(T x) { return f(x); };
}
fun id<U>(U a) -> U { return a; }
fun main() -> int@ { return apply(id)(0); }
```

In the first application `apply(id)` we do not yet know that `T` should be bound to `int`. The instantiation needs to be delayed until the second application `apply(id)(0)`. In general, each application contributes to the system of inequalities that needs to be solved to instantiate the generic function. In ML^{F}, the return type of each application encodes a partial system of inequalities. The inequalities are recorded in the types as lower bounds on type parameters. The following is an example of such a type.

```
fun<U> where { fun<T>(T)->T ≤ U } (U) -> U
```

Postponing instantiation is not as important in $\mathcal{G}$ because functions take multiple parameters and currying is seldom used.

Removal of the arrow rule means that, in some circumstances, the programmer would have to wrap a function inside another function before passing the function as an argument.

Restrict the language to predicative polymorphism Another alternative is to restrict the language so that only monotypes (non-generic types) may be used as the type arguments in an instantiation. This approach is used in by Odersky and Läufer [45] and also by Peyton Jones and Shields [46]. However, this approach reduces the expressiveness of the language for the sake of the convenience of implicit instantiation.

Restrict the language to second-class polymorphism Restricting the language of types to disallow polymorphic types nested inside other types is another way to make the subtyping problem decidable. With this restriction the subtyping problem is solved by normal unification. Languages such as SML and Haskell 98 use this approach. Like the restriction to predicative polymorphism, this approach reduces the expressiveness of the language for the sake of implicit instantiation (and type inference). However, there are many motivating use cases for first-class polymorphism [47], so throwing out first-class polymorphism is not our preferred alternative.

Use a semi-decision procedure Yet another alternative is to use a semi-decision procedure for the subtyping problem. The advantage of this approach is that it allows implicit instantiation to work in more situations, though it is not clear whether this extra flexibility is needed in practice. The down side is that there are instances of the subtyping problem where the procedure diverges and never returns with a solution.

3.6. *Model lookup (constraint satisfaction)*

The basic idea behind model lookup is simple although some of the details are a bit complicated. Consider the following program containing a generic function `foo` with a requirement for `C<T>`.

```
concept C<T> { };
model C<int> { };

fun foo<T> where { C<T> } (T x) -> T { return x; }

fun main() -> int@ {
  return foo(0);// lookup model C<int>
}
```

At the call `foo(0)`, the compiler deduces the binding `T=int` and then seeks to satisfy the `where` clause, with `int` substituted for `T`. In this case the constraint `C<int>` must be satisfied. In the scope of the call `foo(0)` there is a model definition for `C<int>`, so the constraint is satisfied. We call `C<int>` the *model head*.

3.6.1. *Lexical scoping of models*

The design choice to look for models in the lexical scope of the instantiation is an important choice for $\mathcal{G}$, and differentiates it from both Haskell and the concept extension for C++. This

Fig. 12. Intentionally overlapping models.

```
module A {
  model Monoid<int> {
    fun binary_op(int x, int y) -> int@ { return x + y; }
    fun identity_elt() -> int@ { return 0; }
  };
  fun sum<Iter>(Iter first, Iter last) -> int {
    return accumulate(first, last);
  }
}

module B {
  model Monoid<int> {
    fun binary_op(int x, int y) -> int@ { return x * y; }
    fun identity_elt() -> int@ { return 1; }
  };
  fun product<Iter>(Iter first, Iter last) -> int {
    return accumulate(first, last);
  }
}
```

choice improves the modularity of $\mathcal{G}$ by preventing model declarations in separate modules from accidentally conflicting with one another.

For example, in Fig. 12 we create `sum` and `product` functions in modules `A` and `B` respectively by instantiating `accumulate` in the presence of different model declarations. This example would not type check in Haskell, even if the two instance declarations were to be placed in different modules, because instance declarations implicitly leak out of a module when anything in the module is used by another module. This example would be illegal in C++0X concept extension because 1) model definitions must appear in the same namespace as their concept, and 2) if placed in the same namespace, the two model definitions would violate the one-definition-rule.

It is also quite possible for separately developed modules to include model definitions that accidentally overlap. In $\mathcal{G}$, this is not a problem, as the model definitions will each apply within their own module. Model definitions may be explicitly imported from one module to another. The syntax for modules and import declarations is shown below. An interesting extension would be parameterized modules, but we leave that for future work.

```
decl ← module mid { decl ... }  // module
     | scope mid = scope;  // scope alias
     | import scope.cid<type,...>;  // import model
     | public: decl ...  // public region
     | private: decl ...  // private region
```

3.6.2. *Constrained models*

In $\mathcal{G}$, a model definition may itself be parameterized and the type parameters constrained by a `where` clause. Fig. 13 shows a typical example of a parameterized model. The model definition

Fig. 13. Example of parameterized model definition.

```
concept Comparable<T> {
  fun operator==(T,T)->bool@;
};
model Comparable<int> { };

struct list<T> { /*...*/ };

model <T> where { Comparable<T> }
Comparable< list<T> > {
  fun operator==(list<T> x, list<T> y) -> bool@ { /*...*/ }
};

fun generic_foo<C> where { Comparable<C> } (C a, C b) -> bool@
  { return a == b; }

fun main() -> int@ {
  let l1 = @list<int>(); let l2 = @list<int>();
  generic_foo(l1,l2);
  return 0;
}
```

in the example says that for any type T, list<T> is a model of Comparable if T is a model of Comparable. Thus, a model definition is like an inference rule or a Horn clause [48] in logic programming. For example, a model definition of the form

```
model <T1,...,Tn> where { P1, ..., Pn }
Q { ... };
```

corresponds to the Horn clause:

$$(P_1 \text{ and } \dots \text{ and } P_n) \text{ implies } Q$$

The model definitions from the example in Fig. 13 could be represented in Prolog with the following two rules:

```
comparable(int).
comparable(list(T)) :- comparable(T).
```

The algorithm for model lookup is essentially a logic programming engine: it performs unification and backward chaining (similar to how instance lookup is performed in Haskell). Unification is used to determine when the head of a model definition matches. For example, in Fig. 13, in the call to generic_foo the constraint Comparable< list<int> > needs to be satisfied. There is a model definition for Comparable< list<T> > and unification of list<int> and list<T> succeeds with the type assignment T $=$ int. However, we have not yet satisfied Comparable< list<int> > because the where clause of the parameterized model must also be satisfied. The model lookup algorithm therefore proceeds recursively and tries to satisfy Comparable<int>, which in this case is trivial. This process is called *backward chaining*: it starts with a goal (a constraint to be satisfied) and then applies matching rules (model definitions) to reduce the goal into subgoals. Eventually the subgoals are reduced to facts (model definitions without a `where` clause) and the process is complete. As is typical of Prolog implementations, $\mathcal{G}$ processes subgoals in a depth-first manner.

It is possible for multiple model definitions to match a constraint. When this happens the most specific model definition is used, if one exists. Otherwise the program is ill-formed. We say that definition A is a *more specific model* than definition B if the head of A is a substitution instance of the head of B and if the `where` clause of B implies the `where` clause of A. In this context, implication means that for every constraint c in the `where` clause of A, c is satisfied in the current environment augmented with the assumptions from the `where` clause of B.

$\mathcal{G}$ places very few restrictions on the form of a model definition. The only restriction is that all type parameters of a model must appear in the head of the model. That is, they must appear in the type arguments to the concept being modeled. For example, the following model definition is ill formed because of this restriction.

```
concept C<T> { };
model <T,U> C<T> { }; // ill formed, U is not in an argument to C
```

This restriction ensures that unifying a constraint with the model head always produces assignments for all the type parameters.

Horn clause logic is by nature powerful enough to be Turning-complete. For example, it is possible to express general recursive functions. The program in Fig. 14 computes the Ackermann function at compile time by encoding it in model definitions. This power comes at a price: determining whether a constraint is satisfied by a set of model definitions is in general undecidable. Thus, model lookup is not guaranteed to terminate and programmers must take some care in writing model definitions. We could restrict the form of model definitions to achieve decidability however there are two reasons not to do so. First, restrictions would complicate the specification of $\mathcal{G}$ and make it harder to learn. Second, there is the danger of ruling out useful model definitions.

3.7. *Improved error messages*

In the introduction we showed how users of generic libraries in C++ are plagued by hard to understand error messages. The introduction of concepts and where clauses in $\mathcal{G}$ solves this problem. The following is the same misuse of the `stable_sort` function, but this time written in $\mathcal{G}$.

```
4  fun main() -> int@{
5    let v = @list<int>();
6    stable_sort(begin(v), end(v));
7    return 0;
8  }
```

In contrast to long C++ error message (Fig. 1), in $\mathcal{G}$ we get the following:

```
test/stable_sort_error.hic:6:
In application stable_sort(begin(v), end(v)),
Model MutableRandomAccessIterator<mutable_list_iter<int>>
needed to satisfy requirement, but it is not defined.
```

Fig. 14. The Ackermann function encoded in model definitions.

```
struct zero { };
struct suc<n> { };
concept Ack<x,y> { type result; };

model <y> Ack<zero,y> { type result = suc<y>; };

model <x> where { Ack<x, suc<zero> > }
Ack<suc<x>, zero> { type result = Ack<x, suc<zero> >.result; };

model <x,y> where { Ack<suc<x>,y>, Ack<x, Ack<suc<x>,y>.result > }
Ack< suc<x>,suc<y> > {
  type result = Ack<x, Ack<suc<x>,y>.result >.result;
};

fun foo(int) { }
fun main() -> int@ {
  type two = suc< suc<zero> >; type three = suc<two>;
  foo(@Ack<two,three>.result());
  // error: Type (suc<suc<suc<suc<suc<suc<suc<suc<suc<zero>>>>>>>>>)
  // does not match type (int)
}
```

A related problem that plagues authors of generic C++ libraries is that type errors often go unnoticed during library development. Again, this is because C++ delays type checking templates until instantiation. One of the reasons for such type errors is that the implementation of a template is not consistent with its documented type requirements.

This problem is directly addressed in $\mathcal{G}$: the implementation of a generic function is type-checked with respect to its `where` clause, independently of any instantiations. Thus, when a generic function successfully compiles, it is guaranteed to be free of type errors and the implementation is guaranteed to be consistent with the type requirements in the `where` clause.

Interestingly, while implementing the STL in $\mathcal{G}$, the type checker caught several errors in the STL as defined in C++. One such error was in `replace_copy`. The implementation below was translated directly from the GNU C++ Standard Library, with the `where` clause matching the requirements for `replace_copy` in the C++ Standard [49].

```
fun replace_copy<Iter1,Iter2, T>
where { InputIterator<Iter1>, Regular<T>, EqualityComparable<T>,
        OutputIterator<Iter2, InputIterator<Iter1>.value>,
        OutputIterator<Iter2, T>,
        EqualityComparable2<InputIterator<Iter1>.value,T> }
(Iter1@ first, Iter1 last, Iter2@ result, T old, T neu) -> Iter2@ {
  for ( ; first != last; ++first)
    result << *first == old ? neu : *first;
  return result;
}
```

The $\mathcal{G}$ compiler gives the following error message:

```
stl/sequence_mutation.hic:203:
The two branches of the conditional expression must have the
same type or one must be coercible to the other.
```

This is a subtle bug, which explains why it has gone unnoticed for so long. The type requirements say that both the value type of the iterator and `T` must be writable to the output iterator, but the requirements do not say that the value type and `T` are the same type, or coercible to one another.

3.8. *Generic classes, structs, and unions*

The syntax for generic classes, structs, and unions is defined below. The grammar variable *clid* is for class, struct, and union names.

```
decl ← class clid polyhdr { classmem ... };
decl ← struct clid polyhdr { mem ... };
decl ← union clid polyhdr { mem ... };
mem ← type id;
classmem ← mem
         | polyhdr clid(type pass [id], ...) { stmt ... }
         | ~clid() { stmt ... }
polyhdr ← [<tyid,...>] [where { constraint, ... }]
```

Classes consist of data members, constructors, and a destructor. There are no member functions; normal functions are used instead. Data encapsulation (`public`/`private`) is specified at the module level instead of inside the class. Class, struct, and unions are used as types using the syntax below. Such a type is well-formed if the type arguments are well-formed and if the requirements in its where clause are satisfied.

```
type ← clid[<type, ...>]
```

3.9. *Type equality*

There are several language constructions in $\mathcal{G}$ that make it difficult to decide when two types are equal. Generic functions complicate type equality because the names of the type parameters do not matter. So, for example, the following two function types are equal:

```
fun<T>(T)->T = fun<U>(U)->U
```

The order of the type parameters does matter (because a generic function may be explicitly instantiated) so the following two types are not equal.

```
fun<S,T>(S,T)->T ≠ fun<T,S>(S,T)->T
```

Inside the scope of a generic function, type parameters with different names are assumed to be different types (this is a conservative assumption). So, for example, the following program is ill formed because variable `a` has type `S` whereas function `f` is expecting an argument of type `T`.

```
fun foo<S, T>(S a, fun(T)->T f) -> T { return f(a); }
```

Associated types and same-type constraints also affect type equality. First, if there is a model definition in the current scope such as:

```
model C<int> { type bar = bool; };
```

then we have the equality `C<int>.bar` = `bool`.

Inside the scope of a generic function, same-type constraints help determine when two types are equal. For example, the following version of `foo` is well formed:

```
fun foo_1<T, S> where { T == S } (fun(T)->T f, S a) -> T { return f(a); }
```

There is a subtle difference between the above version of `foo` and the following one. The reason for the difference is that same-type constraints are checked after type argument deduction.

```
fun foo_2<T>(fun(T)->T f, T a) -> T { return f(a); }

fun id(double x) -> double { return x; }

fun main() -> int@ {
  foo_1(id, 1.0); // ok
  foo_1(id, 1); // error: Same type requirement violated, double != int
  foo_2(id, 1.0); // ok
  foo_2(id, 1); // ok
}
```

In the first call to `foo_1` the compiler deduces `T=double` and `S=double` from the arguments `id` and `1.0`. The compiler then checks the same-type constraint `T == S`, which in this case is satisfied. For the second call to `foo_1`, the compiler deduces `T=double` and `S=int` and then the same-type constraint `T == S` is not satisfied. The first call to `foo_2` is straightforward. For the second call to `foo_2`, the compiler deduces `T=double` from the type of `id` and the argument `1` is implicitly coerced to `double`.

Type equality is a *congruence relation*, which means several things. First it means type equality is an *equivalence relation*, so it is reflexive, transitive, and symmetric. Thus, for any types ρ, σ, and τ we have

– $\tau = \tau$
– $\sigma = \tau$ implies $\tau = \sigma$
– $\rho = \sigma$ and $\sigma = \tau$ implies $\rho = \tau$

For example, the following function is well formed:

```
fun foo<R,S,T> where { R == S, S == T}
(fun(T)->S f, R a) -> T { return f(a); }
```

The type expression `R` (the type of `a`) and the type expression `T` (the parameter type of `f`) both denote the same type.

The second aspect of type equality being a congruence is that it propagates in certain ways with respect to type constructors. For example, if we know that `S` = `T` then we also know that `fun(S)->S` = `fun(T)->T`. Similarly, if we have defined a generic struct such as:

```
struct bar<U> { };
```

then `S` = `T` implies `bar<S>` = `bar<T>`. The propagation of equality also goes in the other direction. For example, `bar<S>` = `bar<T>` implies that `S` = `T`. The congruence extends to associated types. So `S` = `T` implies `C<S>.bar` = `C<T>.bar`. However, for associated types, the propagation does not go in the reverse direction. So `C<S>.bar` = `C<T>.bar` does not imply that `S` = `T`. For example, given the model definitions

```
model C<int> { type bar = bool; };
model C<float> { type bar = bool; };
```

we have `C<int>.bar` = `C<float>.bar` but this does not imply that `int` = `float`.

Like type parameters, associated types are in general assumed to be different from one another. So the following program is ill-formed:

```
concept C<U> { type bar; };
fun foo<S, T> where { C<S>, C<T> } (C<S>.bar a, fun(C<T>.bar)->T f) -> T
{ return f(a); }
```

The next program is also ill formed.

```
concept D<U> { type bar; type zow; };
fun foo<T> where { D<T> } (D<T>.bar a, fun(D<T>.zow)->T f) -> T
{ return f(a); }
```

In the compiler for $\mathcal{G}$ we use the congruence closure algorithm by Nelson and Oppen [50] to keep track of which types are equal. The algorithm is efficient: $O(n \log n)$ time complexity on average, where n is the number of types. It has $O(n^2)$ time complexity in the worst case. This can be improved by instead using the Downey-Sethi-Tarjan algorithm which is $O(n \log n)$ in the worst case [51].

3.10. *Function overloading and concept-based overloading*

Multiple functions with the same name may be defined and static overload resolution is performed to decide which function to invoke at a particular call site. The resolution depends on the argument types and on the model definitions in scope. When more than one overload may be called, the most specific overload is called if one exists. The basic overload resolution rules are based on those of C++.

In the following simple example, the second `foo` is called.

```
fun foo() -> int@ { return -1; }
fun foo(int x) -> int@ { return 0; }
fun foo(double x) -> int@ { return -1; }
fun foo<T>(T x) -> int@ { return -1; }

fun main() -> int@ { return foo(3); }
```

The first `foo` has the wrong number of arguments, so it is immediately dropped from consideration. The second and fourth are given priority over the third because they can exactly match the argument type `int` (for the fourth, type argument deduction results in `T=int`), whereas the third `foo` requires an implicit coercion from `int` to `double`. The second `foo` is favored over the fourth because it is more specific.

A function f is a *more specific overload* than function g if g is callable from f but not vice versa. A function g is *callable from* function f if you could call g from inside f, forwarding all the parameters of f as arguments to g, without causing a type error. More formally, if f has type `fun<`$\overline{t_f}$`>where`$C_f(\overline{\sigma_f})$`->`τ_f and g has type `fun<`$\overline{t_g}$`>where`$C_g(\overline{\sigma_g})$`->`τ_g then g is callable from f if

$$\overline{\sigma_f} \leq [\overline{t_g}/\overline{\rho}]\overline{\sigma_g} \text{ and } C_f \text{ implies } [\overline{t_g}/\overline{\rho}]C_g$$

for some $\overline{\rho}$.

In general there may not be a most specific overload in which case the program is ill-formed. In the following example, both `foo`'s are callable from each other and therefore neither is more specific.

```
fun foo(double x) -> int@ { return 1; }
```

Fig. 15. The advance algorithms using concept-based overloading.

```
fun advance<Iter> where { InputIterator<Iter> }
(Iter! i, InputIterator<Iter>.difference@ n) {
  for (; n != zero(); --n)
    ++i;
}
fun advance<Iter> where { BidirectionalIterator<Iter> }
(Iter! i, InputIterator<Iter>.difference@ n) {
  if (zero() < n)
    for (; n != zero(); --n)
      ++i;
  else
    for (; n != zero(); ++n)
      --i;
}
fun advance<Iter> where { RandomAccessIterator<Iter> }
(Iter! i, InputIterator<Iter>.difference@ n) {
  i = i + n;
}
```

```
fun foo(float x) -> int@ { return -1; }
fun main() -> int@ { return foo(3); }
```

In the next example, neither foo is callable from the other so neither is more specific.

```
fun foo<T>(T x, int y) -> int@ { return 1; }
fun foo<T>(int x, T y) -> int@ { return -1; }
fun main() -> int@ { return foo(3, 4); }
```

In Section 2.5 we showed how to accomplish concept-based overloading of several versions of advance using the tag dispatching idiom in C++. Fig. 15 shows three overloads of advance implemented in $\mathcal{G}$. The signatures for these overloads are the same except for their where clauses. The concept BidirectionalIterator is a refinement of InputIterator, so the second version of advance is more specific than the first. The concept RandomAccessIterator is a refinement of BidirectionalIterator, so the third advance is more specific than the second.

The code in Fig. 16 shows two calls to advance. The first call is with an iterator for a singly-linked list. This iterator is a model of InputIterator but not RandomAccessIterator; the overload resolution chooses the first version of advance. The second call to advance is with a pointer which is a RandomAccessIterator so the second version of advance is called.

Concept-based overloading in $\mathcal{G}$ is entirely based on static information available during the type checking and compilation of the call site. This presents some difficulties when trying to resolve to optimized versions of an algorithm from within another generic function. Section **??** discusses the issues that arise and presents an idiom that ameliorates the problem.

3.11. *Function expressions*

The following is the syntax for function expressions and function types.

Fig. 16. Example calls to advance and overload resolution.

```
use "slist.g";
use "basic_algorithms.g"; // for copy
use "iterator_functions.g"; // for advance
use "iterator_models.g"; // for iterator models for int*

fun main() -> int@ {
  let sl = @slist<int>();
  push_front(1, sl); push_front(2, sl);
  push_front(3, sl); push_front(4, sl);
  let in_iter = begin(sl);
  advance(in_iter, 2); // calls version 1, linear time

  let rand_iter = new int[4];
  copy(begin(sl), end(sl), rand_iter);
  advance(rand_iter, 2);   // calls version 3, constant time

  if (*in_iter == *rand_iter) return 0;
  else return -1;
}
```

The body of a function expression may be either a sequence of statements enclosed in braces or a single expression following a colon. The return type of a function expression is deduced from the return statements in the body, or from the single expression.

The following example computes the sum of an array using `for_each` and a function expression. [2]

```
fun main() -> int@ {
  let n = 8;
  let a = new int[n];
  for (let i = 0; i != n; ++i)
    a[i] = i;
  let sum = 0;
  for_each(a, a + n, fun(int x) p=&sum { *p = *p + x; });
  return sum - (n * (n-1))/2;
}
```

The expression

```
fun(int x) p=&sum { *p = *p + x; }
```

creates a function object. The body of a function expression is not lexically scoped, so a direct use of `sum` in the body would be an error. The initialization `p=&sum` declares a data member inside the function object with type `int*` and copy constructs the member with the address `&sum`.

[2] Of course, the `accumulate` function is the appropriate algorithm for this computation, but then the example would not demonstrate the use of function expressions.

The primary motivation for non-lexically scoped function expressions is to keep the design close to C++ so that function expressions can be directly compiled to function objects in C++. However, this design has some drawbacks as we discovered while porting the STL to $\mathcal{G}$.

Most STL implementations implement two separate versions of `find_subsequence`, one written in terms of `operator==` and the other in terms of a function object. The version using `operator==` could be written in terms of the one that takes a function object, but it is not written that way. The original reason for this was to improve efficiency, but with with a modern optimizing compiler there should be no difference in efficiency: all that is needed to erase the difference is some simple inlining. The $\mathcal{G}$ implementation we write the `operator==` version of `find_subsequence` in terms of the higher-order version. The following code shows how this is done and is a bit more complicated than we would have liked.

```
fun find_subsequence<Iter1,Iter2>
where { ForwardIterator<Iter1>, ForwardIterator<Iter2>,
        ForwardIterator<Iter1>.value == ForwardIterator<Iter2>.value,
        EqualityComparable<ForwardIterator<Iter1>.value> }
(Iter1 first1, Iter1 last1, Iter2 first2, Iter2 last2) -> Iter1@@
{
  type T = ForwardIterator<Iter1>.value;
  let cmp = model EqualityComparable<T>.operator==;
  return find_subsequence(first1, last1, first2, last2,
                          fun(T a,T b) c=cmp: c(a, b));
}
```

It would have been simpler to write the function expression as

```
fun(T a, T b): a == b
```

However, this is an error in $\mathcal{G}$ because the `operator==` from the `EqualityComparable<..>` requirement is a local name, not a global one, and is therefore not in scope for the body of the function expression. The workaround is to store the comparison function as a data member of the function object. The expression

```
model EqualityComparable<T>.operator==
```

accesses the `operator==` member from the model of `EqualityComparable` for type `T`.

Examples such as these are a convincing argument that lexical scoping should be allowed in function expressions, and the next generation of $\mathcal{G}$ will support this feature.

3.12. *First-class polymorphism*

In the introduction we mentioned that $\mathcal{G}$ is based on System F. One of the hallmarks of System F is that it provides first class polymorphism. That is, polymorphic objects may be passed to and returned from functions. This is in contrast to the ML family of languages, where polymorphism is second class. In Section 3.5 we discussed how the restriction to second-class polymorphism simplifies type argument deduction, reducing it to normal unification. However, we prefer to retain first-class polymorphism and use the somewhat more complicated variant of unification from ML^{F}.

One of the reasons to retain first-class polymorphism is to retain the expressiveness of function objects in C++. A function object may have member function templates and may therefore by used polymorphically. The following program is a simple use of first-class polymorphism in $\mathcal{G}$. Note that `f` is applied to arguments of different types.

Fig. 17. Some STL Algorithms in $\mathcal{G}$.

```
fun find<Iter> where { InputIterator<Iter> }
(Iter@ first, Iter last,
 fun(InputIterator<Iter>.value)->bool@ pred) -> Iter@ {
  while (first != last and not pred(*first)) ++first;
  return first;
}
fun find<Iter> where { InputIterator<Iter>,
    EqualityComparable<InputIterator<Iter>.value> }
(Iter@ first, Iter last, InputIterator<Iter>.value value) -> Iter@ {
  while (first != last and not (*first == value)) ++first;
  return first;
}
fun remove<Iter> where { MutableForwardIterator<Iter>,
    EqualityComparable<InputIterator<Iter>.value> }
(Iter@ first, Iter last, InputIterator<Iter>.value value) -> Iter@ {
  first = find(first, last, value);
  let i = @Iter(first);
  return first == last ? first : remove_copy(++i, last, first, value);
}
```

```
fun foo(fun<T>(T)->T f) -> int@ { return f(1) + d2i(f(-1.0)); }
fun id<T>(T x) -> T { return x; }
fun main() -> int@ { return foo(id); }
```

4. Analysis of $\mathcal{G}$ and the STL

In this section we analyze the interdependence of the language features of $\mathcal{G}$ and generic library design in light of implementing the STL. A primary goal of generic programming is to express algorithms with minimal assumptions about data abstractions, so we first look at how the generic functions of $\mathcal{G}$ can be used to accomplish this. Another goal of generic programming is efficiency, so we investigate the use of function overloading in $\mathcal{G}$ to accomplish automatic algorithm selection. We conclude this section with a brief look at implementing generic containers and adaptors in $\mathcal{G}$.

4.1. *Algorithms*

Fig. 17 depicts a few simple STL algorithms implemented using generic functions in $\mathcal{G}$. The STL provides two versions of most algorithms, such as the overloads for `find` in Fig. 17. The first version is higher-order, taking a predicate function as its third parameter while the second version relies on `operator==`. Functions are first-class in $\mathcal{G}$, so the higher-order version is straightforward to express. As is typical in the STL, there is a high-degree of internal reuse: `remove` uses `remove_copy` and and `find`.

Fig. 18. The STL Iterator Concepts in $\mathcal{G}$ (Part I).

```
concept InputIterator<X> {
  type value;
  type difference;
  refines EqualityComparable<X>;
  refines Regular<X>;
  require SignedIntegral<difference>;
  fun operator*(X) -> value@;
  fun operator++(X!) -> X!;
};
concept OutputIterator<X,T> {
  refines Regular<X>;
  fun operator<<(X!, T) -> X!;
};
concept ForwardIterator<X> {
  refines DefaultConstructible<X>;
  refines InputIterator<X>;
  fun operator*(X) -> value;
};
concept MutableForwardIterator<X> {
  refines ForwardIterator<X>;
  refines OutputIterator<X,value>;
  require Regular<value>;
  fun operator*(X) -> value!;
};
```

4.2. *Iterators*

Figures 18 and 19 show the STL iterator hierarchy as represented in $\mathcal{G}$. Required operations are expressed in terms of function signatures, and associated types are expressed with a nested `type` requirement. The refinement hierarchy is established with the `refines` clauses and nested model requirements with `require`. The semantic invariants and complexity guarantees of the iterator concepts are not expressible in $\mathcal{G}$ as they are beyond the scope of its type system.

4.3. *Automatic Algorithm Selection*

To realize the generic programming efficiency goals, $\mathcal{G}$ provides mechanisms for automatic algorithm selection. The following code shows two overloads for `copy`. (We omit the third overload to save space.) The first version is for input iterators and the second for random access, which uses an integer counter thereby allowing some compilers to better optimize the loop. The two signatures are the same except for the `where` clause.

```
fun copy<Iter1,Iter2> where { InputIterator<Iter1>,
     OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1@ first, Iter1 last, Iter2@ result) -> Iter2@ {
  for (; first != last; ++first) result << *first;
```

Fig. 19. The STL Iterator Concepts in $\mathcal{G}$ (Part II).

```
concept BidirectionalIterator<X> {
  refines ForwardIterator<X>;
  fun operator--(X!) -> X!;
};
concept MutableBidirectionalIterator<X> {
  refines BidirectionalIterator<X>;
  refines MutableForwardIterator<X>;
};
concept RandomAccessIterator<X> {
  refines BidirectionalIterator<X>;
  refines LessThanComparable<X>;
  fun operator+(X, difference) -> X@;
  fun operator-(X, difference) -> X@;
  fun operator-(X, X) -> difference@;
};
concept MutableRandomAccessIterator<X> {
  refines RandomAccessIterator<X>;
  refines MutableBidirectionalIterator<X>;
};
```

```
  return result;
}
fun copy<Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1@ first, Iter1 last, Iter2@ result) -> Iter2@ {
  for (n = last - first; n > zero(); --n, ++first) result << *first;
  return result;
}
```

The use of dispatching algorithms such as copy inside other generic algorithms is challenging because overload resolution is based on the surrogate models from the where clause and not on models defined for the instantiating type arguments. (This rule is needed for separate type checking and compilation). Thus, a call to an overloaded function such as copy may resolve to a non-optimal overload. Consider the following implementation of merge. The Iter1 and Iter2 types are required to model InputIterator and the body of merge contains two calls to copy.

```
fun merge<Iter1,Iter2,Iter3>
where { InputIterator<Iter1>, InputIterator<Iter2>,
        LessThanComparable<InputIterator<Iter1>.value>,
        InputIterator<Iter1>.value == InputIterator<Iter2>.value,
        OutputIterator<Iter3, InputIterator<Iter1>.value> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
   -> Iter3@ { ...
  return copy(first2, last2, copy(first1, last1, result));
}
```

This merge function always calls the slow version of copy even though the actual iterators may be random access. In C++, with tag dispatching, the fast version of copy is called because the overload resolution occurs after template instantiation. However, C++ does not have separate type checking for templates.

To enable dispatching for copy, the type information at the instantiation of merge must be carried into the body of merge (suppose it is instantiated with a random access iterator). This can be done with a combination of concept and model declarations. First, define a concept with a single operation that corresponds to the algorithm.

```
concept CopyRange<I1,I2> {
  fun copy_range(I1,I1,I2) -> I2@;
};
```

Next, add a requirement for this concept to the type requirements of merge and replace the calls to copy with the concept operation copy_range.

```
fun merge<Iter1,Iter2,Iter3>
where { ..., CopyRange<Iter2,Iter3>, CopyRange<Iter1,Iter3> }
(Iter1@ first1, Iter1 last1, Iter2@ first2, Iter2 last2, Iter3@ result)
   -> Iter3@ { ...
  return copy_range(first2, last2, copy_range(first1, last1, result));
}
```

The final step of the idiom is to create parameterized model declarations for CopyRange. The where clauses of the model definitions match the where clauses of the respective overloads for copy. In the body of each copy_range there is a call to copy which will resolve to the appropriate overload.

```
model <Iter1,Iter2> where { InputIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2@
    { return copy(first, last, result); }
};
model <Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2@
    { return copy(first, last, result); }
};
```

A call to merge with a random access iterator will use the second model to satisfy the requirement for CopyRange. Thus, when copy_range is invoked inside merge, the fast version of copy is called. A nice property of this idiom is that calls to generic algorithms need not change. A disadvantage of this idiom is that the interface of the generic algorithms becomes more complex.

4.4. *Containers*

The containers of the STL are implemented in $\mathcal{G}$ using polymorphic classes. Fig. 20 shows an excerpt of the doubly-linked list container in $\mathcal{G}$. As usual, a dummy sentinel node is used in the implementation. With each STL container comes iterator types that translate between the uniform iterator interface and data-structure specific operations. Fig. 20 shows the list_iterator

Fig. 20. Excerpt from a doubly-linked list container in $\mathcal{G}$.

```
struct list_node<T> where { Regular<T>, DefaultConstructible<T> } {
  list_node<T>* next; list_node<T>* prev; T data;
};
class list<T> where { Regular<T>, DefaultConstructible<T> } {
  list() : n(new list_node<T>()) { n->next = n; n->prev = n; }
  ~list() { ... }
  list_node<T>* n;
};
class list_iterator<T> where { Regular<T>, DefaultConstructible<T> } {
  ... list_node<T>* node;
};
fun operator*<T> where { Regular<T>, DefaultConstructible<T> }
(list_iterator<T> x) -> T { return x.node->data; }

fun operator++<T> where { Regular<T>, DefaultConstructible<T> }
(list_iterator<T>! x) -> list_iterator<T>!
    { x.node = x.node->next; return x; }

fun begin<T> where { Regular<T>, DefaultConstructible<T> }
(list<T> l) -> list_iterator<T>@
    { return @list_iterator<T>(l.n->next); }

fun end<T> where { Regular<T>, DefaultConstructible<T> }
(list<T> l) -> list_iterator<T>@ { return @list_iterator<T>(l.n); }
```

which implements `operator*` in terms of `x.node->data` and implements `operator++` by performing the assignment `x.node = x.node->next`.

Not shown in Fig. 20 is the implementation of the mutable iterator for `list` (the `list_iterator` provides read-only access). The definitions of the two iterator types are nearly identical, the only difference is that `operator*` returns by read-only reference for the constant iterator whereas it returns by read-write reference for the mutable iterator. The code for these two iterators should be reused but $\mathcal{G}$ does not yet have a language mechanism for this kind of reuse.

In C++ this kind of reuse can be expressed using the Curiously Recurring Template Pattern (CRTP) [52] and by parameterizing the base iterator class on the return type of `operator*`. This approach can not be used in $\mathcal{G}$ because the parameter passing mode may not be parameterized. Further, the semantics of polymorphism in $\mathcal{G}$ does not match the intended use here, we want to *generate* code for the two iterator types at library construction time. A separate *generative* mechanism is needed to complement the generic features of $\mathcal{G}$. As a temporary solution, we used the m4 macro system to factor the common code from the iterators. The following is an excerpt from the implementation of the iterator operators.

```
define(‘forward_iter_ops’,
‘fun operator*<T> where { Regular<T>, DefaultConstructible<T> }
($1<T> x) -> T $2 { return x.node->data; } ...’)
forward_iter_ops(list_iterator, &) /* readonly */
forward_iter_ops(mutable_list_iter, !) /* readwrite */
```

4.5. *Adaptors*

The `reverse_iterator` class is a representative example of an STL adaptor.

```
class reverse_iterator<Iter>
  where { Regular<Iter>, DefaultConstructible<Iter> }
{
  reverse_iterator(Iter base) : curr(base) { }
  reverse_iterator(reverse_iterator<Iter> other) : curr(other.curr) { }
  Iter curr;
};
```

The `Regular` requirement on the underlying iterator is needed for the copy constructor and `DefaultConstructible` is needed for the default constructor. This adaptor flips the direction of traversal of the underlying iterator, which is accomplished with the following `operator*` and `operator++`. There is a call to `operator--` on the underlying `Iter` type we need to include the requirement for *Bidirectional Iterator*.

```
fun operator*<Iter> where { BidirectionalIterator<Iter> }
(reverse_iterator<Iter> r) -> BidirectionalIterator<Iter>.value
    { let tmp = @Iter(r.curr); return *--tmp; }

fun operator++<Iter> where { BidirectionalIterator<Iter> }
(reverse_iterator<Iter>! r) -> reverse_iterator<Iter>!
    { --r.curr; return r; }
```

Polymorphic model definitions are used to establish that `reverse_iterator` is a model of the iterator concepts, as we discussed in Section 3.2.

5. The Boost Graph Library

A group of us at the Open Systems Lab performed a comparative study of language support for generic programming [17]. We evaluated a half dozen modern programming languages by implementing a subset of the Boost Graph Library [13] in each language. We implemented a family of algorithms associated with breadth-first search, including Dijkstra's single-source shortest paths [53] and Prim's minimum spanning tree algorithms [54]. This section extends the previous study to include $\mathcal{G}$. We give a brief overview of the BGL, describe the implementation of the BGL in $\mathcal{G}$, and compare the results to those in our earlier study [17].

5.1. *An overview of the BGL graph search algorithms*

Figure 21 depicts some graph search algorithms from the BGL, their relationships, and how they are parameterized. Each large box represents an algorithm and the attached small boxes represent type parameters. An arrow labeled `<uses>` from one algorithm to another specifies that one algorithm is implemented using the other. An arrow labeled `<models>` from a type parameter to an unboxed name specifies that the type parameter must model that concept. For example, the breadth-first search algorithm has three type parameters: `G`, `C`, and `Vis`. Each of these has requirements: G must model the *Vertex List Graph* and *Incidence Graph* concepts, C must model the *Read/Write Map* concept, and Vis must model the *BFS Visitor* concept. The breadth-first search algorithm is implemented using the graph search algorithm.

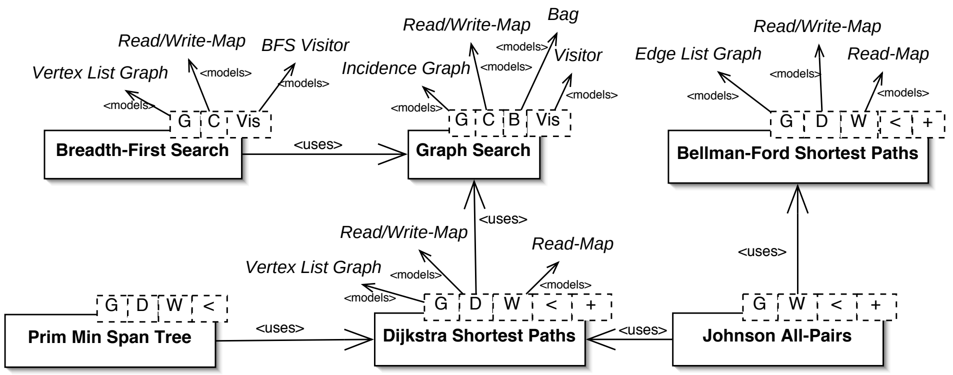

Fig. 21. Graph algorithm parameterization and reuse within the Boost Graph Library. Arrows for redundant models relationships are not shown. For example, the type parameter G of breadth-first search must also model *Incidence Graph* because breadth-first search uses graph search.

The core algorithm of this library is graph search, which traverses a graph and performs user-defined operations at certain points in the search. The order in which vertices are visited is controlled by a type argument, B, that models the *Bag* concept. This concept abstracts a data structure with insert and remove operations but no requirements on the order in which items are removed. When B is bound to a FIFO queue, the traversal order is breadth-first. When it is bound to a priority queue based on distance to a source vertex, the order is closest-first, as in Dijkstra's single-source shortest paths algorithm. Graph search is also parameterized on actions to take at event points during the search, such as when a vertex is first discovered. This parameter, Vis, must model the *Visitor* concept (which is not to be confused with the Visitor design pattern). The graph search algorithm also takes a type parameter C for mapping each vertex to a color and C must model the *Read/Write Map* concept. The colors are used as markers to keep track of the progression of the algorithm through the graph.

The *Read Map* and *Read/Write Map* concepts represent variants of an important abstraction in the graph library: the *property map*. In practice, graphs represent domain-specific entities. For example, a graph might depict the layout of a communication network, its vertices representing endpoints and its edges representing direct links. In addition to the number of vertices and the edges between them, a graph may associate values to its elements. Each vertex of a communication network graph might have a name and each edge a maximum transmission rate. Some algorithms require access to domain information associated with the graph representation. For example, Prim's minimum spanning tree algorithm requires "weight" information associated with each edge in a graph. Property maps provide a convenient implementation-agnostic means of expressing, to algorithms, relations between graph elements and domain-specific data. Some graph data structures directly contain associated values with each node; others use external associative data structures to implement these relationships. Interfaces based on property maps work equally well with both representations.

The graph algorithms are all parameterized on the graph type. Breadth-first search takes a type parameter G, which must model two concepts, *Incidence Graph* and *Vertex List Graph*. The

Incidence Graph concept defines an interface for accessing out-edges of a vertex. *Vertex List Graph* specifies an interface for accessing the vertices of a graph in an unspecified order. The Bellman-Ford shortest paths algorithm [55] requires a model of the *Edge List Graph* concept, which provides access to all the edges of a graph.

That graph capabilities are partitioned among three concepts illustrates generic programming's emphasis on minimal algorithm requirements. The Bellman-Ford shortest paths algorithm requires of a graph only the operations described by the *Edge List Graph* concept. Breadth-first search, in contrast, requires the functionality of two separate concepts. By partitioning the functionality of graphs, each algorithm can be used with any data type that meets its minimum requirements. If the three fine-grained graph concepts were replaced with one monolithic concept, each algorithm would require more from its graph type parameter than necessary and would thus unnecessarily restrict the set of types with which it could be used.

The graph library design is suitable for evaluating generic programming capabilities of languages because its implementation involves a rich variety of generic programming techniques. Most of the algorithms are implemented using other library algorithms: breadth-first search and Dijkstra's shortest paths use graph search, Prim's minimum spanning tree algorithm uses Dijkstra's algorithm, and Johnson's all-pairs shortest paths algorithm [56] uses both Dijkstra's and Bellman-Ford shortest paths. Furthermore, type parameters for some algorithms, such as the `G` parameter to breadth-first search, must model multiple concepts. In addition, the algorithms require certain relationships between type parameters. For example, consider the graph search algorithm. The `C` type argument, as a model of *Read/Write Map*, is required to have an associated key type. The `G` type argument is required to have an associated vertex type. Graph search requires that these two types be the same.

As in our earlier study, we focus the evaluation on the interface of the breadth-first search algorithm and the infrastructure surrounding it, including concept definitions and an example use of the algorithm.

5.2. *Implementation in $\mathcal{G}$*

So far we have implemented breadth-first search and Dijkstra's single-source shortest paths in $\mathcal{G}$. This required defining several of the graph and property map concepts and an implementation of the `adjacency_list` class, a FIFO queue, and a priority queue.

The interface for the breadth-first search algorithm is straightforward to express in $\mathcal{G}$. It has three type parameters: the graph type `G`, the color map type `C`, and the visitor type `Vis`. The requirements on the type parameters are expressed with a `where` clause, using concepts that we describe below. In the interface of `breadth_first_search`, associated types and same-type constraints play an important role in accurately tracking the relationships between the graph type, its vertex descriptor type, and the color property map.

```
type Color = int;
let black = 0;
let gray  = 1;
let white = 2;

fun breadth_first_search<G, C, Vis>
  where { IncidenceGraph<G>, VertexListGraph<G>,
          ReadWritePropertyMap<C>,
          PropertyMap<C>.key == IncidenceGraph<G>.vertex_descriptor,
          PropertyMap<C>.value == Color,
          BFSVisitor<Vis,G> }
(G g, IncidenceGraph<G>.vertex_descriptor@ s, C c, Vis vis) { /* ... */ }
```

Figure 22 shows the definition of several graph concepts in $\mathcal{G}$. The Graph concept requires the associated types vertex_descriptor and edge_descriptor and some basic functionality for those types such as copy construction and equality comparison. This concept also includes the source and target functions. The Graph concept serves to factor common requirements out of the IncidenceGraph and VertexListGraph concepts.

The IncidenceGraph concept introduces the capability to access out-edges of a vertex. The access is provided by the out_edge_iterator associated type. The requirements for the out-edge iterator are slightly more than the standard InputIterator concept and slightly less than the ForwardIterator concept. The out-edge iterator must allow for multiple passes but dereferencing an out-edge iterator need not return a reference (for example, it may return by-value instead). Thus we define the following new concept to express these requirements.

```
concept MultiPassIterator<Iter> {
    refines DefaultConstructible<Iter>;
    refines InputIterator<Iter>;
    // semantic requirement: allow multiple passes through the range
};
```

In Figure 22, the IncidenceGraph concept uses same-type constraints to require that the value type of the iterator to be the same type as the edge_descriptor. The VertexListGraph concepts adds the capability of traversing all the vertices in the graph using the associated vertex_iterator.

Figure 23 shows the implementation of a graph in terms of a vector of singly-linked lists. Vertex descriptors are integers and edge descriptors are pairs of integers. The out-edge iterator is implemented with the vg_out_edge_iter class whose implementation is shown in Figure 24. The basic idea behind this iterator is to provide a different view of the list of target vertices, making it appear as a list of source-target pairs.

The property map concepts are defined in Figure 25. The ReadWritePropertyMap is a refinement of the ReadablePropertyMap concept, which requires the get function, and the WritablePropertyMap concept, which requires the put function. Both of these concepts refine the PropertyMap concept which includes the associated key and value types.

Figure 26 shows the definition of the BFSVisitor concept. This concept is naturally expressed as a multi-parameter concept because the visitor and graph types are independent: a particular visitor may be used with many different concrete graph types and vice versa. The use of refines for Graph in BFSVisitor is somewhat odd, require would be more natural, but the refinement provides direct (and convenient) access to the vertex and edge descriptor types. An alternative would be use to require and some type aliases, but type aliases have not yet been added to concept definitions.

Figure 27 presents an example use of the breadth_first_search function to output vertices in breadth-first order. To do so, the test_vis visitor overrides the function discover_vertex; empty implementations of the other visitor functions are provided by default_bfs_visitor. A graph is constructed using the AdjacencyList class, and then breadth_first_search is called.

Fig. 22. Graph concepts in $\mathcal{G}$.

```
concept Graph<G> {
    type vertex_descriptor;
    require DefaultConstructible<vertex_descriptor>;
    require Regular<vertex_descriptor>;
    require EqualityComparable<vertex_descriptor>;

    type edge_descriptor;
    require DefaultConstructible<edge_descriptor>;
    require Regular<edge_descriptor>;
    require EqualityComparable<edge_descriptor>;

    fun source(edge_descriptor, G) -> vertex_descriptor@;
    fun target(edge_descriptor, G) -> vertex_descriptor@;
};
concept IncidenceGraph<G> {
    refines Graph<G>;

    type out_edge_iterator;
    require MultiPassIterator<out_edge_iterator>;
    edge_descriptor == InputIterator<out_edge_iterator>.value;

    fun out_edges(vertex_descriptor, G)
        -> pair<out_edge_iterator, out_edge_iterator>@;
    fun out_degree(vertex_descriptor, G) -> int@;
};
concept VertexListGraph<G> {
    refines Graph<G>;

    type vertex_iterator;
    require MultiPassIterator<vertex_iterator>;
    vertex_descriptor == InputIterator<vertex_iterator>.value;

    fun vertices(G) -> pair<vertex_iterator, vertex_iterator>@;
    fun num_vertices(G) -> int@;
};
```

6. Related Work

There is a long history of programming language support for polymorphism, dating back to the 1970s [20, 57, 58, 59]. An early precursor to $\mathcal{G}$'s concept feature can be seen in CLU's type set feature [58]. Type sets differ from concepts in that they rely on structural conformance whereas concepts use nominal conformance established by a `model` definition. Also, $\mathcal{G}$ provides a means for composing concepts via refinement whereas CLU does not provide a means for composing type sets. Finally, CLU does not provide support for associated types.

In mathematics, the notion of algebraic structure is equivalent to $\mathcal{G}$'s concept, and has been in

Fig. 23. Implementation of a graph with a vector of lists.

```
fun source(pair<int,int> e, vector< slist<int> >) -> int@
  { return e.first; }
fun target(pair<int,int> e, vector< slist<int> >) -> int@
  { return e.second; }

model Graph< vector< slist<int> > > {
  type vertex_descriptor = int;
  type edge_descriptor = pair<int,int>;
};

fun out_edges(int src, vector< slist<int> > G)
    -> pair<vg_out_edge_iter, vg_out_edge_iter>@ {
  return make_pair(@vg_out_edge_iter(src, begin(G[src])),
                   @vg_out_edge_iter(src, end(G[src])));
}
fun out_degree(int src, vector< slist<int> > G) -> int@
  { return size(G[src]); }

model IncidenceGraph< vector< slist<int> > > {
  type out_edge_iterator = vg_out_edge_iter;
};

fun vertices(vector< slist<int> > G)
  -> pair<counting_iter,counting_iter>@
  { return make_pair(@counting_iter(0), @counting_iter(size(G))); }
fun num_vertices(vector< slist<int> > G) -> int@ { return size(G); }

model VertexListGraph< vector< slist<int> > > {
  type vertices_size_type = int;
  type vertex_iterator = counting_iter;
};
```

use for a very long time [60].

Type classes The concept feature in $\mathcal{G}$ is heavily influenced by the type class feature of Haskell [61], with its nominal conformance and explicit model definitions. However, $\mathcal{G}$'s support for associated types, same type constraints, and concept-based overloading is novel. Also, $\mathcal{G}$'s type system is fundamentally different from Haskell's: it is based on System F [20, 57] instead of Hindley-Milner type inference [59]. This difference has some repercussions. In $\mathcal{G}$ there is more control over the scope of concept operations because `where` clauses introduce concept operations into the scope of the body. This difference allows Haskell to infer type requirements but induces the restriction that two type classes in the same module may not have operations with the same name. A difference we discussed in Section 3.6.1 is that in $\mathcal{G}$, overlapping models may coexist in separate scopes but still be used in the same program, whereas in Haskell overlapping models may not be used in the same program. Haskell performed quite well in our comparative study of support for generic programming [17]. However, we pointed out that Haskell was

Fig. 24. Out-edge iterator for the vector of lists.

```
class vg_out_edge_iter {
  vg_out_edge_iter() { }
  vg_out_edge_iter(int src, slist_iterator<int> iter)
    : src(src), iter(iter) { }
  vg_out_edge_iter(vg_out_edge_iter x)
    : iter(x.iter), src(x.src) { }
  slist_iterator<int> iter;
  int src;
};
fun operator=(vg_out_edge_iter! me, vg_out_edge_iter other)
  -> vg_out_edge_iter!
  { me.iter = other.iter; me.src = other.src; return me; }
model DefaultConstructible<vg_out_edge_iter> { };
model Regular<vg_out_edge_iter> { };

fun operator==(vg_out_edge_iter x, vg_out_edge_iter y) -> bool@
  { return x.iter == y.iter; }
fun operator!=(vg_out_edge_iter x, vg_out_edge_iter y) -> bool@
  { return x.iter != y.iter; }
model EqualityComparable<vg_out_edge_iter> { };

fun operator*(vg_out_edge_iter x) -> pair<int,int>@
  { return make_pair(x.src, *x.iter); }
fun operator++(vg_out_edge_iter! x) -> vg_out_edge_iter!
  { ++x.iter; return x; }
model InputIterator<vg_out_edge_iter> {
  type value = pair<int,int>;
  type difference = ptrdiff_t;
};
model MultiPassIterator<vg_out_edge_iter> { };
```

missing support for associated types and work to remedy this has been reported in [22, 23].

Wehr, Lammel, and Thiemann[62] have proposed extending Java with generalized interfaces, which bear a close resemblance to $\mathcal{G}$'s concepts and Haskell's type classes, but add the capability of run-time dispatch using existential quantification. ($\mathcal{G}$ currently provides only universal quantification, although programmers can workaround this limitation with an tricky encoding [63]).

Signatures and functors A rough analogy can be made between SML signatures [36] and $\mathcal{G}$ concepts, and between ML structures and $\mathcal{G}$ models. However, there are significant differences. Functors are module-level constructs and therefore provide a more coarse-grained mechanism for parameterization than do generic functions. More importantly, functors require explicit instantiation with a structure, thereby making their use more heavyweight than generic functions in F^{G}, which perform automatic lookup of the required model or instance. The associated types and same-type constraints of $\mathcal{G}$ are roughly equivalent to types nested in ML signatures and to type sharing respectively. We reuse some implementation techniques from ML such as a union/find-based algorithm for deciding type equality [64]. There are numerous other languages

Fig. 25. Property map concepts in $\mathcal{G}$.

```
concept PropertyMap<Map> {
    type key;
    type value;
};
concept ReadablePropertyMap<Map> {
    refines PropertyMap<Map>;
    fun get(Map, key) -> value;
};
concept WritablePropertyMap<Map> {
    refines PropertyMap<Map>;
    fun put(Map, key, value);
};
concept ReadWritePropertyMap<Map>  {
    refines ReadablePropertyMap<Map>;
    refines WritablePropertyMap<Map>;
};
```

Fig. 26. Breadth-first search visitor concept.

```
concept BFSVisitor<Vis, G> {
    refines Regular<Vis>;
    refines Graph<G>;

    fun initialize_vertex(Vis v, vertex_descriptor d, G g) {}
    fun discover_vertex(Vis v, vertex_descriptor d, G g) {}
    fun examine_vertex(Vis v, vertex_descriptor d, G g) {}
    fun examine_edge(Vis v, edge_descriptor d, G g) {}
    fun tree_edge(Vis v, edge_descriptor d, G g) {}
    fun non_tree_edge(Vis v, edge_descriptor d, G g) {}
    fun gray_target(Vis v, edge_descriptor d, G g) {}
    fun black_target(Vis v, edge_descriptor d, G g) {}
    fun finish_vertex(Vis v, vertex_descriptor d, G g) {}
};
```

with parameterized modules [65, 66, 67] that require explicit instantiation with a structure.

Recently, Dreyer, Harper, Chakravarty, and Keller proposed an extension of SML signatures/-functors, call modular type classes [68], that provides many of the benefits of Haskell type classes such as implicit instantiation and instance passing. The design for modular type classes differs from concepts in $\mathcal{G}$ primarily in that it supports the convenience of type inference at the price of disallowing overlapping instances in a given scope and first-class polymorphism.

Subtype-bound polymorphism Less closely related to $\mathcal{G}$ are languages based on subtype-bounded polymorphism [69] such as Java, C#, and Eiffel. We found subtype-bounded poly-

Fig. 27. Example use of the BFS generic function.

```
struct test_vis { };
fun discover_vertex<G>(test_vis, int v, G g) { printf("%d ", v); }

model <G> where { Graph<G>,  Graph<G>.vertex_descriptor == int }
BFSVisitor<test_vis, G> { };

fun main() -> int@ {
  let n = 7;
  let g = @vector< slist<int> >(n);
  push_front(1, g[0]); push_front(4, g[0]);
  push_front(2, g[1]); push_front(3, g[1]);
  push_front(4, g[3]); push_front(6, g[3]);
  push_front(5, g[4]);

  let src = 0;
  let color = new Color[n];
  for (let i = 0; i != n; ++i)
    color[i] = white;
  breadth_first_search(g, src, color, @test_vis());
  return 0;
}
```

morphism less suitable for generic programming and refer the reader to [17] for an in-depth discussion.

Row variable polymorphism OCaml's object types[37, 70] and polymorphism over row variables provide fairly good support for generic programming. However, OCaml lacks support for associated types so it suffers from clutter due to extra type parameters in generic functions. PolyTOIL [71], with its match-bound polymorphism, provides similar support for generic programming as OCaml but also lacks associated types.

Virtual types One of the proposed solutions for dealing with binary methods and associated types in object-oriented languages is *virtual types*, that is, the nesting of abstract types in interfaces and type definitions within classes or objects. The beginning of this line of research was the *virtual patterns* feature of the BETA language [72]. Patterns are a generalization of classes, objects, and procedures. An adaptation of virtual patterns to object-oriented classes, called *virtual classes*, was created by Madsen and Moller-Pedersen [73] and an adaptation for Java was created by Thorup [74]. These early designs for virtual types were not statically type safe, but relied on dynamic type checking. However, a statically type safe version was created by Torgersen [75]. A statically type safe version of BETA's virtual patterns was developed for the gbeta language of Ernst [76, 77]; the Scala programming language also includes type safe virtual types [78, 79].

7. Conclusion

This article presents a new programming language named $\mathcal{G}$ that is designed to meet the needs of large-scale generic libraries. We demonstrated this with an implementation of the Standard

Template Library (STL) and the Boost Graph Library (BGL). We were able to implement all of the abstractions in the STL and BGL in a straightforward manner. Further, $\mathcal{G}$ is particularly well-suited for the development of reusable components due to its support of modular type checking and separate compilation. $\mathcal{G}$'s strong type system provides support for the independent validation of components and $\mathcal{G}$'s system of concepts and constraints allows for rich interactions between components without sacrificing encapsulation. The language features present in $\mathcal{G}$ promise to increase programmer productivity with respect to the development and use of generic components.

Acknowledgments

We thank Ronald Garcia, Jeremiah Willcock, Doug Gregor, Jaakko Järvi, Dave Abrahams, Dave Musser, and Alexander Stepanov for many discussions and collaborations that informed this work. This work was supported by NSF grants EIA-0131354 and CCF-0702362, and by a grant from the Lilly Endowment.

References

[1] Randell, B.: Software engineering in 1968. In: ICSE '79: Proceedings of the 4th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1979) 1–10

[2] Frederick P. Brooks, J.: The Mythical Man-Month: Essays on Softw. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1978)

[3] McIlroy, M.D.: Mass-produced software components. In Buxton, J.M., Naur, P., Randell, B., eds.: Proceedings of Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering. (1969) 138–155 `http://www.cs.dartmouth.edu/~doug/components.txt`.

[4] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley, Reading, MA (2002)

[5] Kapur, D., Musser, D.R., Stepanov, A.: Operators and algebraic structures. In: Proc. of the Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, ACM (1981)

[6] Musser, D.R., Stepanov, A.A.: Generic programming. In Gianni, P.P., ed.: Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings. Volume 358 of Lecture Notes in Computer Science., Berlin, Springer Verlag (1989) 13–25

[7] Musser, D.R., Stepanov, A.A.: A library of generic algorithms in Ada. In: Using Ada (1987 International Ada Conference), New York, NY, ACM SIGAda (1987) 216–225

[8] Kershenbaum, A., Musser, D., Stepanov, A.: Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute (1988)

[9] Stroustrup, B.: Parameterized types for C++. In: USENIX C++ Conference. (1988)

[10] Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project (1994)

[11] Austern, M.H.: Generic programming and the STL: Using and extending the C++ Standard Template Library. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)

[12] Köthe, U.: Reusable Software in Computer Vision. In: Handbook on Computer Vision and Applications. Volume 3. Acadamic Press (1999)

[13] Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

[14] Boissonnat, J.D., Cazals, F., Da, F., Devillers, O., Pion, S., Rebufat, F., Teillaud, M., Yvinec, M.: Programming with CGAL: the example of triangulations. In: Proceedings of the fifteenth annual symposium on Computational geometry, ACM Press (1999) 421–422

[15] Pitt, W.R., Williams, M.A., Steven, M., Sweeney, B., Bleasby, A.J., Moss, D.S.: The bioinformatics template library: generic components for biocomputing. Bioinformatics **17** (2001) 729–737

[16] Troyer, M., Todo, S., Trebst, S., and, A.F.: (ALPS: Algorithms and Libraries for Physics Simulations) `http://alps.comp-phys.org/`.

[17] Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A comparative study of language support for generic programming. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 115–134

[18] Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An extended comparative study of language support for generic programming. Journal of Functional Programming **17** (2007) 145–205

[19] Siek, J., Lumsdaine, A.: Essential language support for generic programming. In: PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2005) 73–84

[20] Girard, J.Y.: Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur. Thèse de doctorat d'état, Université Paris VII, Paris, France (1972)

[21] Reynolds, J.C.: Types, abstraction and parametric polymorphism. In Mason, R.E.A., ed.: Information Processing 83, Amsterdam, Elsevier Science Publishers B. V. (North-Holland) (1983) 513–523

[22] Chakravarty, M.M.T., Keller, G., Peyton Jones, S., Marlow, S.: Associated types with class. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2005) 1–13

[23] Chakravarty, M.M.T., Keller, G., Peyton Jones, S.: Associated type synonyms. In: ICFP '05: Proceedings of the International Conference on Functional Programming, New York, NY, USA, ACM Press (2005) 241–253

[24] Jarvi, J., Gregor, D., Willcock, J., Lumsdaine, A., Siek, J.G.: Algorithm specialization in generic programming - challenges of constrained generics in C++. In: PLDI '06: Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2006)

[25] Siek, J.G.: A Language for Generic Programming. PhD thesis, Indiana University (2005)

[26] Siek, J., Lumsdaine, A.: Language requirements for large-scale generic libraries. In: GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering. (2005) accepted for publication.

[27] Gregor, D., Järvi, J., Siek, J.G., Reis, G.D., Stroustrup, B., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06). (2006)

[28] Peyton Jones, S., Jones, M., Meijer, E.: Type classes: an exploration of the design space. In: Proceedings of the Second Haskell Workshop. (1997)

[29] Jazayeri, M., Loos, R., Musser, D., Stepanov, A.: Generic Programming. In: Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany (1998)
[30] Austern, M.: (draft) technical report on standard library extensions. Technical Report N1711=04-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2004)
[31] Silicon Graphics, Inc.: SGI Implementation of the Standard Template Library. (2004) `http://www.sgi.com/tech/stl/`.
[32] Musser, D.R.: Introspective sorting and selection algorithms. Software Practice and Experience **27** (1997) 983–993
[33] Hoare, C.A.R.: Algorithm 64: Quicksort. Communications of the ACM **4** (1961) 321
[34] Myers, N.C.: Traits: a new and useful template technique. C++ Report (1995)
[35] Järvi, J., Willcock, J., Lumsdaine, A.: Algorithm specialization and concept constrained genericity. In: Concepts: a Linguistic Foundation of Generic Programming, Adobe Systems (2004)
[36] Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press (1990)
[37] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml Documentation and User's Manual. (2003)
[38] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, C., Scheifler, B., Snyder, A.: CLU reference manual. Technical Report LCS-TR-225, MIT, Cambridge, MA, USA (1979)
[39] Ditchfield, G.J.: Overview of Cforall. University of Waterloo (1996)
[40] Magnusson, B.: Code reuse considered harmful. Journal of Object-Oriented Programming **4** (1991)
[41] Järvi, J., Stroustrup, B., Gregor, D., Siek, J.: Decltype and auto. Technical Report N1478=03-0061, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003) `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf`.
[42] Mitchell, J.C.: Polymorphic type inference and containment. Information and Computation **76** (1988) 211–249
[43] Tiuryn, J., Urzyczyn, P.: The subtyping problem for second-order types is undecidable. Information and Computation **179** (2002) 1–18
[44] Le Botlan, D., Rémy, D.: MLF: Raising ML to the power of System-F. In: Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, ACM Press (2003) 27–38
[45] Odersky, M., Läufer, K.: Putting type annotations to work. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1996) 54–67
[46] Jones, S.P., Shields, M.: Practical type inference for arbitrary-rank types. submitted to the Journal of Functional Programming (2004)
[47] chieh Shan, C.: Sexy types in action. SIGPLAN Notices **39** (2004) 15–22
[48] Horn, A.: On sentences which are true of direct unions of algebras. Journal of Symbolic Logic **16** (1951) 14–21
[49] International Organization for Standardization: ISO/IEC 14882:1998: Programming languages — C++, Geneva, Switzerland (1998)
[50] Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27** (1980) 356–364
[51] Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. Journal of the ACM (JACM) **27** (1980) 758–771

[52] Coplien, J.: Curiously recurring template patterns. C++ Report (1995) 24–27

[53] Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271

[54] Prim, R.: Shortest connection networks and some generalizations. Bell System Technical Journal **36** (1957) 1389–1401

[55] Bellman, R.: On a routing problem. Quarterly of Applied Mathematics **16** (1958) 87–90

[56] Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. Journal of the ACM **24** (1977) 1–13

[57] Reynolds, J.C.: Towards a theory of type structure. In Robinet, B., ed.: Programming Symposium. Volume 19 of LNCS., Berlin, Springer-Verlag (1974) 408–425

[58] Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Communications of the ACM **20** (1977) 564–576

[59] Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375

[60] Bourbaki, N.: Elements of Mathematics. Theory of Sets. Springer (1968)

[61] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages, ACM (1989) 60–76

[62] Wehr, S., Lämmel, R., Thiemann, P.: JavaGI: Generalized Interfaces for Java. In: ECOOP 2007, Proceedings. LNCS, Springer-Verlag (2007) 25 pages; To appear.

[63] Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)

[64] MacQueen, D.: An implementation of Standard ML modules. In: Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT, New York, NY, ACM (1988) 212–223

[65] Poll, E., Thompson, S.: The Type System of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NF, UK (1999)

[66] Goguen, J.A., Winker, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In: Applications of Algebraic Specification using OBJ. Cambridge University Press (1992)

[67] : Ada 95 Reference Manual. (1997)

[68] Dreyer, D., Harper, R., Chakravarty, M.M.T., Keller, G.: Modular type classes. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2007) 63–70

[69] Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM Press (1989) 273–280

[70] Rémy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory And Practice of Object Systems **4** (1998) 27–50 A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[71] Bruce, K.B., Schuett, A., van Gent, R.: PolyTOIL: A type-safe polymorphic object-oriented language. In Olthoff, W., ed.: Proceedings of *ECOOP '95*. Number 952 in Lecture Notes in Computer Science, Springer-Verlag (1995) 27–51

[72] Kristensen, B.B., Madsen, O.L., M&#248;ller-Pedersen, B., Nygaard, K.: Abstraction mechanisms in the BETA programming language. In: POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1983) 285–298

[73] Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA '89: Conference proceedings on Object-oriented pro-

gramming systems, languages and applications, New York, NY, USA, ACM Press (1989) 397–406

[74] Thorup, K.K.: Genericity in Java with virtual types. In: ECOOP '97. Volume 1241 of Lecture Notes in Computer Science. (1997) 444–471

[75] Torgersen, M.: Virtual types are statically safe. In: FOOL 5: The Fifth International Workshop on Foundations of Object-Oriented Languages. (1998)

[76] Ernst, E.: gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999)

[77] Ernst, E.: Family polymorphism. In: ECOOP '01. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 303–326

[78] Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proc. ECOOP'03. Springer LNCS (2003)

[79] Odersky, M., al.: An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)